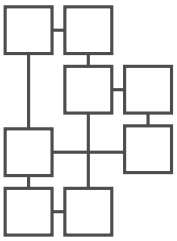
Vom Fachbereich Informatik der Technischen Universität Darmstadt genehmigte Dissertation
zur Erlangung des akademischen Grades Doktor-Ingenieur (Dr.-Ing.)



A Microservice Architecture for the Processing of Large Geospatial Data in the Cloud

M.Sc. Michel Krämer

geboren in Gießen

Referenten der Arbeit:

Prof. Dr. techn. Dieter W. Fellner
Technische Universität Darmstadt

Assoc. Prof. Dr. Jan Boehm
University College London

Tag der Einreichung: 05.09.2017
Tag der mündlichen Prüfung: 07.11.2017

D17 - Darmstadt 2018

Abstract

With the growing number of devices that can collect spatiotemporal information, as well as the improving quality of sensors, the geospatial data volume increases constantly. Before the raw collected data can be used, it has to be processed. Currently, expert users are still relying on desktop-based Geographic Information Systems to perform processing workflows. However, the volume of geospatial data and the complexity of processing algorithms exceeds the capacities of their workstations. There is a paradigm shift from desktop solutions towards the Cloud, which offers virtually unlimited storage space and computational power, but developers of processing algorithms often have no background in computer science and hence no expertise in Cloud Computing.

Our research hypothesis is that a microservice architecture and Domain-Specific Languages can be used to orchestrate existing geospatial processing algorithms, and to compose and execute geospatial workflows in a Cloud environment for efficient application development and enhanced stakeholder experience. We present a software architecture that contains extension points for processing algorithms (or microservices), a workflow management component for distributed service orchestration, and a workflow editor based on a Domain-Specific Language. The main aim is to provide both users and developers with the means to leverage the possibilities of the Cloud, without requiring them to have a deep knowledge of distributed computing. In order to conduct our research, we follow the Design Science Research Methodology. We perform an analysis of the problem domain and collect requirements as well as quality attributes for our architecture. To meet our research objectives, we design the architecture and develop approaches to workflow management and workflow modelling. We demonstrate the utility of our solution by applying it to two real-world use cases and evaluate the quality of our architecture based on defined scenarios. Finally, we critically discuss our results.

Our contributions to the scientific community can be classified into three pillars. We present a scalable and modifiable microservice architecture for geospatial processing that supports distributed development and has a high availability. Further, we present novel approaches to service integration and orchestration in the Cloud as well as rule-based and dynamic workflow management without a priori design-time knowledge. For the workflow modelling we create a Domain-Specific Language that is based on a novel language design method.

Our evaluation results support our hypothesis. The microservice architectural style enables efficient development of a distributed system. The Domain-Specific Language and our approach to service integration enhance stakeholder experience. Our work is a major step within the paradigm shift towards the Cloud and opens up possibilities for future research.

Zusammenfassung

Mit der wachsenden Zahl an Geräten, die spatio-temporale Informationen aufnehmen können sowie immer besser werdenden Sensoren, steigt auch die Menge an Geodaten. Vor der Benutzung müssen die rohen gesammelten Informationen verarbeitet werden. Zurzeit greifen Experten auf desktop-basierte Geographische Informationssysteme zurück, um Prozessierungsworkflows durchzuführen. Allerdings übersteigt das Datenvolumen sowie die Komplexität der Verarbeitungsalgorithmen längst die Kapazität ihrer Workstations. Zurzeit findet ein Paradigmenwechsel von Desktop-Lösungen zur Cloud statt, aber die Entwickler von Prozessierungsalgorithmen sind oft keine Informatiker und haben deshalb wenig Erfahrung im Bereich Cloud-Computing.

Unsere Forschungshypothese ist, dass eine Microservice-Architektur und domänenspezifische Sprachen genutzt werden können, um existierende Algorithmen zu orchestrieren und Workflows für die Prozessierung von Geodaten in der Cloud auszuführen, und damit eine effiziente Anwendungsentwicklung ermöglichen und die Erfahrung von Stakeholdern verbessern. Wir präsentieren eine Softwarearchitektur, die Erweiterungspunkte für Prozessierungsalgorithmen (oder Microservices) enthält, eine Workflow-Management-Komponente für die verteilte Service-Orchestrierung, und einen Workflow-Editor basierend auf einer domänenspezifischen Sprache. Ziel ist es, Benutzern und Entwicklern ohne tiefergehendes Wissen in verteilten Systemen den Zugang zur Cloud zu ermöglichen. Unsere Forschungsmethode basiert auf der Design Science Research Methodology. Wir führen eine Analyse der Problemdomäne durch und sammeln Anforderungen und Qualitätsattribute für unsere Architektur. Um unsere Forschungsziele zu erreichen, entwickeln wir die Architektur sowie Ansätze für Workflow-Management und -Modellierung. Wir stellen den Nutzen unserer Lösung dar, indem wir sie auf zwei praktische Anwendungsfälle anwenden. Außerdem evaluieren wir ihre Qualität anhand von definierten Szenarien. Abschließend führen wir eine kritische Bewertung unserer Ergebnisse durch.

Unsere wissenschaftlichen Beiträge können in drei Bereiche gegliedert werden. Wir präsentieren eine skalierbare und erweiterbare Microservice-Architektur für die Geodatenprozessierung, die eine verteilte Entwicklung ermöglicht sowie eine hohe Verfügbarkeit bietet. Außerdem präsentieren wir neue Ansätze für die Service-Integration und -Orchestrierung in der Cloud sowie regelbasiertes und dynamisches Workflow-Management ohne a priori Wissen im Entwurf. Für die Workflow-Modellierung entwickeln wir eine domänenspezifische Sprache sowie eine neue Methode fürs Sprachdesign.

Die Ergebnisse unserer Arbeit stützen unsere Forschungshypothese. Die Microservice-Architektur ermöglicht eine effiziente Entwicklung eines verteilten Systems. Die domänenspezifische Sprache sowie unser Ansatz zur Service-Integration verbessern die Erfahrung der Stakeholder. Unsere Arbeit stellt einen großen Schritt im Paradigmenwechsel zur Cloud dar und bietet Möglichkeiten für weitere Forschung.

Acknowledgements

This thesis would not have been possible without the support of many people. I would like to thank my colleagues (and friends!) at the Fraunhofer Institute for Computer Graphics Research IGD. This particularly includes Eva Klien who gave me the freedom to pursue my research and to write this thesis. I would also like to thank the people who reviewed my drafts and gave me valuable input (in alphabetical order): Quillon Harpham, Zaheer Khan, Eva Klien, Arjan Kuijper, Thomas Letschert, Joachim Rix, Kamran Soomro, and Evie Stannard. Finally, thanks to Nicolas Paparoditis for the permission to use images from his paper (Paparoditis et al., 2012) for Section 1.8.1.

Notation

Figures in this thesis depicting a software architecture, a specific part of an architecture, or a dynamic structure have been created using the Fundamental Modeling Concepts (FMC) *block diagram* and *petri net* notation (Keller et al., 2002). FMC can be used to describe and communicate complex software architectures with a limited set of symbols and rules.

FMC block diagrams describe the structure and the components of a system. The following list summarises the main elements (see also Figure 1):

- *Stickmen:* Active human actors
- *Rectangles (boxes with straight edges):* Active components that serve a well-defined purpose—e.g. controllers and web services
- *Boxes with round edges:* Passive systems, components channels or storage—e.g. files, databases, and communication channels
- *Arrows and connecting lines:* Access type—read or write (arrows) or both (lines)
- *Circles:* Communication channels with a directed request direction

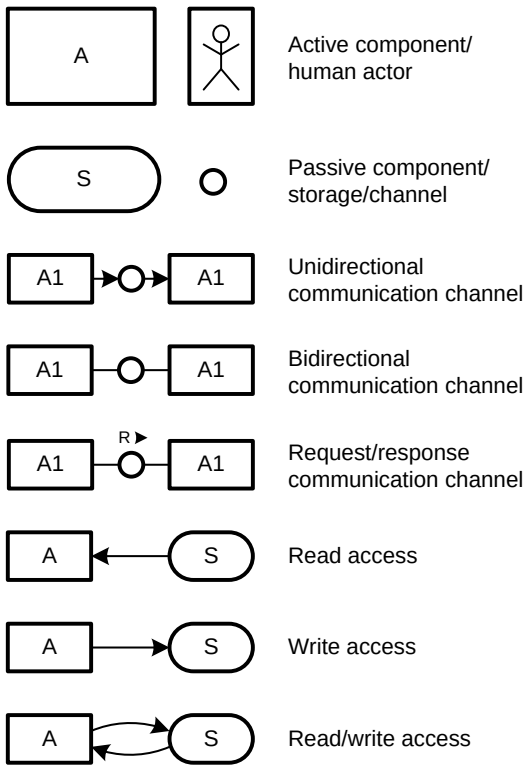


Figure 1 Summary of the elements in an FMC block diagram

FMC petri nets are used to depict the dynamic structures of a system, which means the behaviour of the system and the actions performed by the components. The main elements are (see also Figure 2):

- *Transitions:* An operation, an event or an activity
- *Places*: A control state or a condition
- *Arrows*: Connect places and transitions

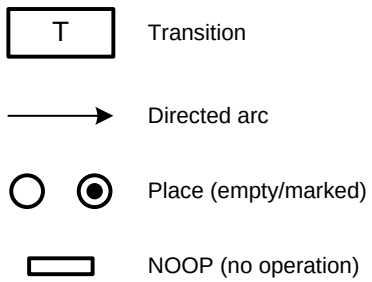


Figure 2 Summary of the elements in an FMC petri net

There are a few other elements in FMC. For a complete overview including examples we refer to the official notation reference (Apfelbacher & Rozinat, 2003).

Table of Contents

1 Introduction

The amount of information that is collected and processed today grows exponentially. It is estimated that by 2025 the global data volume will have reached 163 zettabytes, which is a trillion gigabytes (Reinsel, Gantz, & Rydning, 2017). The main drivers of this growth are social media, mobile devices, the Internet of Things (IoT), and the growing number of sensors built into various devices such as smartphones or (autonomous) cars.

A large part of the produced information can be located in time and place (Vatsavai et al., 2012). This kind of information is called *spatiotemporal data* (or *geospatial data*, *geodata*). For many years, GPS technology has found its way into households, with location sensors built into consumer devices such as navigational systems or smartphones. These devices track their owner's position, record waypoints and routes, and save location information in every photo taken (Goodchild, 2007). In addition, earth observation satellites, as well as airborne laser scanners or terrestrial mobile mapping systems, offer similar data streams. Such devices record hundreds of thousands of samples per second (Cahalane, McCarthy, & McElhinney, 2012) and produce amounts of data ranging from a few GiB up to several TiB in a couple of hours (Paparoditis et al., 2012).

Geospatial data can be of great value for a number of applications. For example, point clouds acquired by earth observation satellites can be used to regularly generate digital terrain models of large areas and to monitor changes in the landscape. This is useful for estimating the risk of landslides or for calculating the hydraulic energy produced by rain water running down steep terrain. In urban areas, geospatial data can be used for multiple use cases related to urban planning, environmental protection or disaster management. Data recorded by mobile mapping systems can be analysed to identify individual objects such as trees and to monitor their biomass for environmental protection.

Before the acquired geospatial data can be used in any of these applications it has to be processed. For example, point clouds generated by earth observation satellites need to be converted to a surface (i.e. triangulated to a digital terrain model), and the data acquired by mobile mapping systems in urban areas needs to be analysed to identify individual objects. The processing should happen in a reasonable amount of time, so that applications can make use of the most up-to-date information. However, there are inherent challenges related to geospatial data processing. Yang et al. (2011) differentiate between four factors of influence: *a)* the high data volume, *b)* the complexity of spatial processing algorithms, *c)* the improving accuracy and better coverage of modern devices, as well as *d)* the growing demand to share data and to concurrently access or process it for various purposes.

Due to this, geospatial data has been recognised as *Big Data* (Kitchin & McArdle, 2016), which means it often exceeds the capacities of current computer systems in terms of available storage, computational power, as well as bandwidth. New distributed computing paradigms such as the Cloud address this issue. The Cloud is scalable, resilient, fault tolerant, and suitable for storing and processing growing amounts of data, while being responsive and centrally accessible. In recent years, it has become one of the major drivers of industry. Since hardware has become rather inexpensive and network connections have become faster—even over long distances—it is now possible to build large, high-performance clusters of commodity computer systems. The nodes in such Clouds can be used in concert to process large amounts of data in a very short time. Additionally, the cost for data storage is so low that Clouds can provide virtually unlimited space.

According to Mell & Grance (2011) from the U.S. National Institute of Standards and Technology (NIST) the Cloud model is composed of three service layers: *Infrastructure as a Service (IaaS)*, *Platform as a Service (PaaS)* and *Software as a Service (SaaS)*. There are a number of vendors offering commercial platforms and targeting at least one of these layers. For example, Amazon Web Services (AWS), Google Cloud Platform or Microsoft Azure offer services in all three layers, whereas Salesforce.com or IBM Bluemix provide PaaS services and target customers who want to deploy their own SaaS solutions. In the geospatial domain large market players have only recently started to make use of the Cloud. Esri, the market leader for geospatial solutions, for example, offer first SaaS applications on AWS and Microsoft Azure. A wider use of the Cloud in the geospatial community is not observable yet, but there is a paradigm shift towards it which will lead to a general acceptance in the coming years.

1.1 Paradigm shift towards GIS in the Cloud

Today, geospatial data is typically managed with desktop-based Geographic Information Systems (GIS) such as Esri ArcGIS or the open-source tool QGIS. The origins of GIS date back to the late 1960s, when the surveying community was faced with novel challenges stemming from the desire to use new sources of data and new techniques to analyse maps, as well as to be able to edit, verify and classify the data (Coppock & Rhind, 1991). The first GIS ran on large mainframe computers controlled by punch-cards. With the advent of the personal computer in the 1980s, Geographic Information Systems became widely accepted, which leveraged the digitisation of the geospatial domain.

One of the first tools available to a broad audience was GRASS GIS, a free software initially developed by a number of federal agencies of the United States as well as private companies, with the aim to create a solution that could manage their growing catalogue of geospatial data sets. GRASS GIS is a modular system that consists of a number of individual command-line programs that can be called subsequently to perform custom spatial processing workflows. In the late 1990s, a graphical user interface was added to GRASS GIS, which allowed users to control the command-line programs and to display their results. At about the same time, Esri launched ArcGIS for Desktop which became the market leader for desktop-based GIS ever since. The introduction of graphical user interfaces in Geographic Information Systems was a major milestone that contributed to their broad success in the market.

Similar to the launch of GIS software and the implementation of graphical user interfaces, the geospatial market is now facing a new paradigm shift from desktop-based GIS to the Cloud. As described above, the Cloud offers many possibilities, in particular for the management of large data sets, but it is not yet widely used in the geospatial market. Although users increasingly face limitations with current solutions and the volume of geospatial data as well as the complexity of the processing algorithms exceed the storage and compute capabilities of their workstations, traditional desktop-based GIS offers a range of functionality that is not yet available in the Cloud.

This not only applies to the number of spatial processing operations and algorithms the solutions offer, but also to the possibility to automate recurring work (or workflows) by creating scripts. For example, ArcGIS and QGIS allow users to create small programs in a general-purpose programming language such as Python. Automating recurring workflows can save time and money, but *a complete solution that offers a functionality similar to desktop-based products as well as the possibility to create workflows for the processing of geospatial data with a user-friendly interface does not exist yet in the Cloud*. In addition, current solutions based on general-purpose programming languages require expertise that users often do not have. Most of them have no background in computer science and do not want to deal with the technical details of workflow execution. In a distributed environment this issue becomes even more complex.

1.2 Processing large geospatial data in the Cloud

The paradigm shift from desktop to the Cloud not only challenges users but also software developers who provide spatial operations and processing algorithms to Geographic Information Systems. A majority of these algorithms are very stable and have been tested in production for many years. However, since the algorithms were initially created for workstations, they are at best multi-threaded but not immediately suitable to be parallelised in a distributed environment such as the Cloud. In fact, most of the algorithms are single-threaded. In order to transfer them to the Cloud and to fully make use of the possibilities in terms of scalability and computational power, the algorithms need to be modified or completely rewritten—e.g. in MapReduce (Dean & Ghemawat, 2008) or a similar programming paradigm for distributed computing. In fact, many types of algorithms cannot be easily mapped and need to be completely redesigned.

Besides software developers in companies producing GIS solutions, there is a large scientific community with researchers who create state-of-the-art algorithms for geospatial processing. These researchers have different backgrounds such as mathematics, physics, photogrammetry, geomatics, geoinformatics, or related sciences. As such they are not computer scientists and have limited knowledge of programming of distributed applications. Executing their algorithms in the Cloud and making use of its full computational power are hard challenges for them. In fact, having to deal with the technicalities and characteristics of Cloud Computing prevents these researchers from focussing on their actual work—i.e. the creation of novel spatial algorithms.

Another challenge stems from the fact that MapReduce and similar programming paradigms allow for creating single distributed algorithms, but not for workflows that consist of a chain of algorithms. Researchers often work together with colleagues from other institutions and try to create processing workflows by combining algorithms they have developed independently. *At present, there is no workflow management system available that specifically targets geospatial data processing in the Cloud and that is flexible enough to be able to orchestrate and parallelise existing processing algorithms.*

1.3 Problem statement

To summarise the challenges described above, we differentiate between two groups of people: *users* of Geographic Information Systems, as well as *developers and researchers* providing spatial operations and processing algorithms.

Users require

- an interface providing them with the means to process arbitrarily large geospatial data sets in the Cloud with the same set of operations and algorithms they know from their desktop-based GIS,
- the possibility to create workflows in order to automate recurring tasks and to execute them in the Cloud, as well as
- a user interface for workflow creation that does not require them to deal with the technical details of distributed computing or the Cloud infrastructure.

Developers and researchers require

- a way to execute their existing algorithms in the Cloud and to use its potential in terms of computing power and scalability, without having to fundamentally modify or re-implement their algorithms,
- an interface that allows them to integrate their algorithms without having to deal with the technical details of distributed computing such as parallelisation, data distribution and fault tolerance, and
- the possibility to orchestrate their algorithms and combine them with those from other parties in order to create complex processing workflows.

1.4 Objectives

In this thesis we aim to create a software architecture that addresses the challenges discussed in the previous sections. The architecture should assist both GIS users and developers in leveraging the possibilities of the Cloud. It should contain interfaces and extension points that allow developers to integrate their processing algorithms. Integration should not require fundamental modifications to the services. Instead, our architecture should be capable of parallelising existing algorithms (even single-threaded ones) and handling issues such as scalability and fault-tolerance without requiring the developers to have a deep knowledge of distributed computing.

Since the architecture should have the potential to replace a desktop GIS and to provide similar functionality in the Cloud, it should be modular so that many developers and researchers can contribute spatial operations and processing algorithms. These developers and researchers may work for various international companies and institutions that provide state-of-the-art components. The possibility to develop software artefacts in a distributed manner and to integrate them at a central place therefore plays an important role for the architecture.

The user interface of our architecture should allow users to create automated processing workflows for recurring tasks. It should be user-centric and hide unnecessary technical details, so that GIS users with no background in computer science can leverage the Cloud and overcome the limitations of their current workstations. Our architecture should be able to interpret the defined workflows and to orchestrate the algorithms contributed by the developers and researchers accordingly. Workflow execution should be scalable and utilise available Cloud resources to process arbitrary volumes of geospatial data.

1.5 Hypothesis and approach

We formulate the following research hypothesis:

> A microservice architecture and Domain-Specific Languages can be used to orchestrate existing geospatial processing algorithms, and to compose and execute geospatial workflows in a Cloud environment for efficient application development and enhanced stakeholder experience.

The *microservice architecture* is a style for designing software architectures where independent and isolated services act in concert to create a larger application. Each service (or *microservice*) runs in its own process and fulfils a defined purpose, similar to the geospatial processing algorithms described above. The architecture we present in this thesis is based on the microservice architectural style. As we will show later, this approach has significant benefits over the Service-Oriented Architecture traditionally used for distributed applications, in particular in terms of isolation of the services, as well as scalability and fault tolerance of the system. In addition, it offers the possibility to align the structure of the system to the organisational structure of the developing team and hence enables independent and distributed development. Since loose coupling is one of the core concepts, a microservice architecture can be easily extended and maintained. In our case this should allow us to reach our goal related to the integration of multiple processing algorithms contributed by distributed teams of developers and researchers and therefore enable efficient application development.

In order to enhance stakeholder experience, we will look at the requirements from users as well as developers. To orchestrate processing algorithms and to enable the execution of geospatial processing workflows, we will implement a component that works similarly to a scientific workflow management system. To integrate existing algorithms (or microservices) into our architecture we will present a novel way to describe the service interfaces in a machine-readable manner. Service execution and parallelisation in the Cloud will happen transparently to the developers who can therefore better focus on the algorithms. Finally, we will create a Domain-Specific Language (DSL) for the definition of workflows. A DSL is a small programming language targeted at a certain application domain. It is easy to understand for users from this domain, because it is based on vocabulary they are familiar with. Our Domain-Specific Language will have just enough elements to define a geospatial workflow. Its limited expressiveness will make it easier to learn and help users avoid common mistakes in distributed computing (such as concurrent write access to the same data set). In order to design the language, we will create our own modelling method which will be based on best practises from software engineering.

1.6 Contributions

The contributions of this thesis to the scientific community are organised in three pillars. We present a *software architecture* that contributes to the area of large geospatial data processing. This architecture contains a workflow management system for distributed *data processing* in the Cloud. *Workflow definition* is based on a Domain-Specific Language that hides the technical details of distributed computing from the users. The individual contributions of these pillars are described in the following in detail.

1.6.1 Architecture

The main contribution of this thesis is our software architecture for the processing of large geospatial data in the Cloud. It has the following major properties:

Scalability. The architecture supports the processing of arbitrarily large volumes of data. It makes use of available Cloud resources and can scale out (horizontally) if new resources are added. In one of the use cases we present later (see Section 1.8) this will allow us to keep given time constraints and to process geospatial data as fast as it is acquired.

Modifiability. Our architecture is based on microservices. These services are loosely coupled and can be developed and deployed independently. This makes the architecture very modular and allows us to integrate various geospatial processing services which contribute to the overall functionality. The microservice architectural style provides good maintainability and helps create a sustainable system.

Development distributability. Distributed teams of developers and researchers with different backgrounds can work independently and create components that can be integrated into our architecture at a central location to build a single application. This enables us to extend the functionality of our system by state-of-the-art algorithms developed by international experts in geospatial processing.

Availability. Microservices are isolated components that run in their own processes and communicate over lightweight protocols. Due to this, our architecture has a high tolerance to the kind of faults that may happen in distributed environments. As we will show, our system is robust and continues to work if individual components fail. This also allows the distributed teams of developers to independently and continuously deploy new versions of their components without affecting system operation.

1.6.2 Processing

The second pillar of our thesis relates to distributed data processing and contributes to the fields of service orchestration and workflow management systems. Our main aim in this regard is to enable developers and researchers to leverage the possibilities of the Cloud for their own geospatial processing algorithms.

Service integration. We present a way to describe service interfaces (through service metadata) which is generic, lightweight, and covers a wide range of cases. This allows developers and researchers to contribute state-of-the-art processing algorithms to our architecture without requiring fundamental modifications.

Service orchestration. Our architecture contains a component called *JobManager* which is a Workflow Management System. It converts user-defined workflows to executable process chains by orchestrating processing services. Based on the service interface descriptions, it is able to discover services and to create valid chains where outputs of services are compatible to the inputs of subsequent services. Service executions are parallelised if possible, without requiring service developers to implement specific features for distributed computing.

Dynamic workflow management. Our system supports dynamic workflows whose configurations can change during execution. We only require a priori runtime knowledge (see Section 3.2.2). Other Workflow Management Systems require a priori design-time knowledge and can only execute static workflows where all variables have to be known before the workflow is started. Some of these systems offer workarounds for dynamic workflows, but we present an integrated approach.

Rule-based workflow execution. Our JobManager employs a rule-based system to convert workflows to process chains. The rules are configurable and can be adapted to various use cases as well as different executing infrastructures. The rules are also responsible for selecting services and data sets. In addition, they generate hints for our scheduler to distribute work to specific compute nodes in order to leverage data locality and to reduce network traffic.

1.6.3 Workflow modelling

The main aim of the third pillar of this thesis is to provide GIS users with the possibility to access the Cloud and to process large geospatial data without a deep knowledge of distributed computing. To this end, we provide a user-centric interface based on a Domain-Specific Language (DSL) which is a lean programming language tailored to a certain application domain. Specifically, we contribute to the scientific community in the following ways:

DSL for workflow modelling. We present a Domain-Specific Language for the processing of geospatial data. The language is modular and targets users from the domains of urban planning and land monitoring. It is easy to learn and—due to its limited expressiveness—prevents users with no IT background from making mistakes common to distributed computing such as concurrent write access to shared resources.

Novel DSL modelling method. In order to create our Domain-Specific Language, we present a novel incremental and iterative modelling method. This method makes use of best practises from software engineering as it encompasses domain analysis and modelling. These actions help identify relevant terms and actions for the Domain-Specific Language and ensure that the language is tailored to the analysed domain.

1.7 Research design

We follow a slight variation of the Design Science Research Methodology (DSRM). We create a solution for a defined problem and evaluate its utility and quality (Hevner, March, Park, & Ram, 2004). DSRM provides a nominal process model for doing Design Science research as well as a mental model for presenting and evaluating research (Peffers, Tuunanen, Rothenberger, & Chatterjee, 2007). Our method comprises the following steps:

1. Problem identification and motivation. Above, we have identified the problem of processing large geospatial data and motivated the creation of a software architecture. In addition, we perform a literature review for each of the three pillars we contribute to in our main Chapters 2, *Architecture*, 3, *Processing*, and 4, *Workflow Modelling*. We compare existing work to our approaches and identify gaps.

2. Define the objectives for a solution. For the major objectives of our research we refer to Section 1.4. Following up on this, we formulate stakeholder requirements as well as quality at-

tributes for our software architecture in Chapter 2, *Architecture*. These requirements are derived from our work in various international research projects as well as our experience from developing large software systems and collaborating, over the last nine years, with domain users from municipalities, regional authorities, federal agencies, and the industry.

3. Design and development. We present our solution in our three main chapters. It consists of *a)* the software architecture and components for *b)* workflow-based data processing and *c)* workflow modelling with Domain-Specific Languages. Each part of the solution has separate scientific contributions embedded in its design (see Section 1.6).

4. Demonstration and evaluation. We carry out experiments based on two real-world use cases to demonstrate that our software architecture provides a solution to the formulated problem. These use cases are introduced in Section 1.8. In Chapter 5, *Evaluation* we perform a quantitative and a qualitative evaluation of our solution based on the formulated stakeholder requirements and quality attributes. We make use of scenarios which describe actors, stimuli, expected outcomes and response measures. We critically reflect each result and discuss strengths and possible weaknesses.

5. Communication. We have communicated our research results in various publications, extended abstracts, posters, and talks. A list of these can be found in Appendix B, *Scientific work*.

1.8 Use cases

According to our research design, we define requirements for our system based on our work in international research projects, the development of large software systems, and the collaboration with domain users over the last years. In order to evaluate our approach and implementation, we specifically focus on two use cases dealing with urban planning and land monitoring. Both use cases were formulated by GIS users within the IQmulus research project. They describe real-world scenarios with actual problems and goals.

IQmulus was a project funded from the 7th Framework Programme of the European Commission, call identifier FP7-ICT-2011-8, under the grant agreement no. 318787, which started in November 2012 and finished in October 2016. The main aim of IQmulus was to create a platform for the fusion and analysis of high-volume geospatial data such as point clouds, coverages and volumetric data sets. One of the major objectives was to automate geospatial processing as much as possible and reduce the amount of human interaction with the platform. In the project we exploited modern Cloud technology in terms of processing power and distributed storage. As shown in Chapter 5, *Evaluation*, we were able to use the results from this thesis successfully in this project.

1.8.1 Use case A: Urban planning

The first use case describes typical tasks in a municipality or mapping authority. The GIS experts working there need to continuously keep cadastral data sets such as 2D maps or 3D city models up to date. They also perform environmental tasks such as monitoring the growth of trees. For this, the GIS experts make use of information from different sources including aerial images and LiDAR point clouds (Light Detection And Ranging) acquired by airborne laser scanning or laser mobile mapping systems (LMMS).

Figure 1.1 shows the STEREOPOLIS II mobile mapping system as it is used by the national mapping agency of France, the Institut Géographique National (IGN), as well as a visualisation

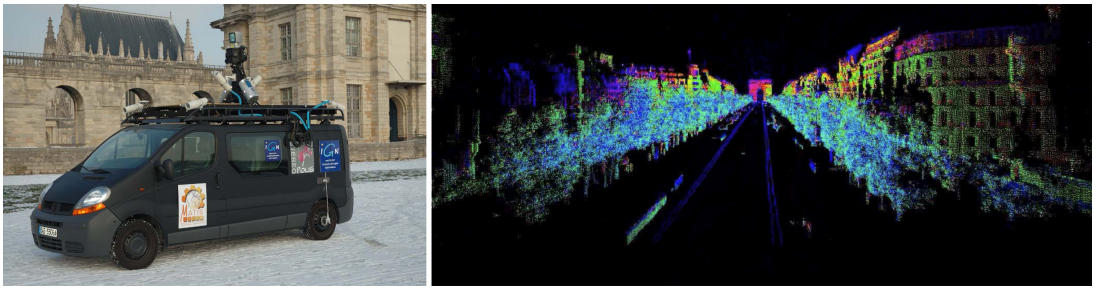


Figure 1.1 The STEREOPOLIS II mobile mapping system by IGN (left) and a 3D point cloud acquired by the two upper RIEGL LiDAR devices (height coloured) over the Champs-Elysées avenue (right). Image source: Paparoditis et al. (2012)

of a large 3D point cloud captured by this system on the Champs-Elysées avenue, Paris, France. The main challenges are the extraction of meaningful information from captured point clouds in an automated way and to handle the data volume and the velocity in which it is acquired. On a typical day of operation, STEREOPOLIS II generates hundreds of millions of points and several terabytes of data (Paparoditis et al., 2012). The average speed of the vehicle is 15 km/h. Within six hours it can cover about 90 linear kilometres. The captured point clouds are unstructured and unclassified. They contain raw geospatial coordinates and timestamps for each collected point. STEREOPOLIS II can be equipped with an image sensor to take panoramic high-definition images and to add colour information to the point clouds.

In the IQmulus project we worked together with end-users from the urban planning domain and identified the following user stories (Belényesi & Kristóf, 2014):

> **User story A.1:** As an urban planner, I want to capture topographic objects (such as cable networks, street edges, urban furniture, traffic lights, etc.) from data acquired by mobile mapping systems (LiDAR point clouds and images) so I can create or update topographic city maps.

> **User story A.2:** As an urban planner, I want to automatically detect individual trees from a LiDAR point cloud in an urban area, so I can monitor growth and foresee pruning work.

> **User story A.3:** As an urban planner, I would like to update my existing 3D city model based on analysing recent LiDAR point clouds.

> **User story A.4:** As an urban planner, I want to provide architects and other urban planners online access to the 3D city model using a simple lightweight web client embedded in any kind of web browser, so that they are able to integrate their projects into the model and share it with decision makers and citizens for communication and project assessment purposes.

Note that user story A.4 describes a specific feature that was requested by users in the IQmulus project. Web-based visualisation of geospatial data is, however, not part of this work. We included this user story because it provides input to one of the examples we present in Chapter 4, *Workflow Modelling* to demonstrate our modelling method for Domain-Specific Languages. Other than that, the user story is not considered any further in this work.

The user stories A.1 to A.3, on the other hand, describe the tasks discussed above. Municipalities and mapping agencies want to keep their data sets such as cadastral maps or 3D city models up to date. In addition, they need to monitor the growth of trees to coordinate pruning work. To this

end, they analyse point clouds to identify building façades and individual objects such as traffic lights or trees. Since the point clouds are so large, the process should be completely automatic. Looking at the visualisation in Figure 1.1, with the human eye we can identify façades, two rows of trees and a couple of street items. If we just consider the vegetation, identifying *individual* trees is, however, very challenging. Doing this in an automated way with a computer is even more so. This is due to the following reasons:

- Trees appear in a variety of sizes and shapes
- They are often only partially visible to the mobile mapping system
- Trees are located at different distances from the road, and may be close to façades, people, cars, street lights, other trees, etc.

There are existing geospatial processing algorithms addressing these issues (Monnier, Vallet, & Soheilian, 2012; Sirmacek & Lindenbergh, 2015). Updating cadastral data sets and monitoring trees are continuous tasks that rely on up-to-date information, but the existing algorithms are very complex and applying them to a large data set can take a long time. The end-users from the IQmulus project reported that analysing the point clouds collected by the STEREOPOLIS II system takes much more time than the data acquisition. For example, a data set collected in the city of Toulouse, France within two hours, comprising more than 1.5 billion points with a total size of about 121 GiB took 52 hours of processing on a workstation that the end-users had access to. Considering that the STEROEPOLIS II system can typically operate for about six hours per day, continuously acquiring more data while the earlier data has not been processed completely reveals a major efficiency bottleneck. Keeping cadastral maps up to date and monitoring tree growth for a whole city is challenging, even on a weekly or monthly basis. The main obstacle of this use case is therefore to process large point clouds faster than they are acquired. In Chapter 5, *Evaluation* we show that this is indeed possible with our architecture.

1.8.2 Use case B: Land monitoring

The Liguria region in the north-west of Italy is a narrow, arch shaped strip of land bordered by the Ligurian sea, the Alps and the Apennine mountains. 65% of the terrain is mountainous, the rest is hilly. Some mountains rise above 2,000 m. The region's orography and its closeness to the sea contribute to the generation of complex hydro-meteorological events. There are a large number of drainage basins (or water catchments) that are connected in a hierarchical pattern (see Figure 1.2). During rainfall, water runs down from the mountains into these basins and subsequently into lower basins until it reaches the sea. This process creates considerable hydraulic energy. Heavy rainfall can cause floods, landslides, and in consequence, major environmental catastrophes. For example, in October 2011 there was an event with more than 468.8 mm of rain falling within 6 hours, with a maximum intensity of 143.4 mm per hour (D'Amato Avanzi, Galanti, Giannecchini, & Bartelletti, 2015). The water flooded three rivers and caused at least 658 shallow landslides. Thirteen people died during this event. The total cost was estimated at 1 Billion Euro. This kind of events occur on a regular basis. Other notable major events happened in November 2011 and two times in 2014 causing many deaths and considerable economic damage.

In order to better prepare against such events, the environmental department of the Liguria region ("Regione Liguria") needs to study orographic precipitation and understand the topography of the mountains in this area. Together with experts from this department, we specified the following user stories (Belényesi & Kristóf, 2014):

> **User story B.1:** As an hydrologist or a geo-morphologist supporting decision makers in civil protection, I want to analyse data measured during critical events to prepare better prediction and monitoring of floods and landslides.

> **User story B.2:** As an hydrologist, I want to study the evolution of measured precipitation data as well as slope deformation from optical images, compute parameters to produce high-quality input for hydrological and mechanical modelling and simulation, and compare the results to reference measurements obtained for flooding events and landslides.

The experts from the environmental department use LiDAR point clouds collected by airborne laser scanners. There are regular flights organised by the Italian Ministry of Environment to keep the data sets up to date and to study the evolution of the terrain over time. One such data set covers the whole Liguria region, has a high resolution and is therefore very large.

In this work we focus on the infrastructure and the parallelisation of the processing algorithms in order to speed up the process. The experts from the environmental department reported that a test on one of their workstations with initial versions of the processing algorithms took several days. In Chapter 5, *Evaluation* we show that, due to our approach, the same process can be performed in about half an hour.

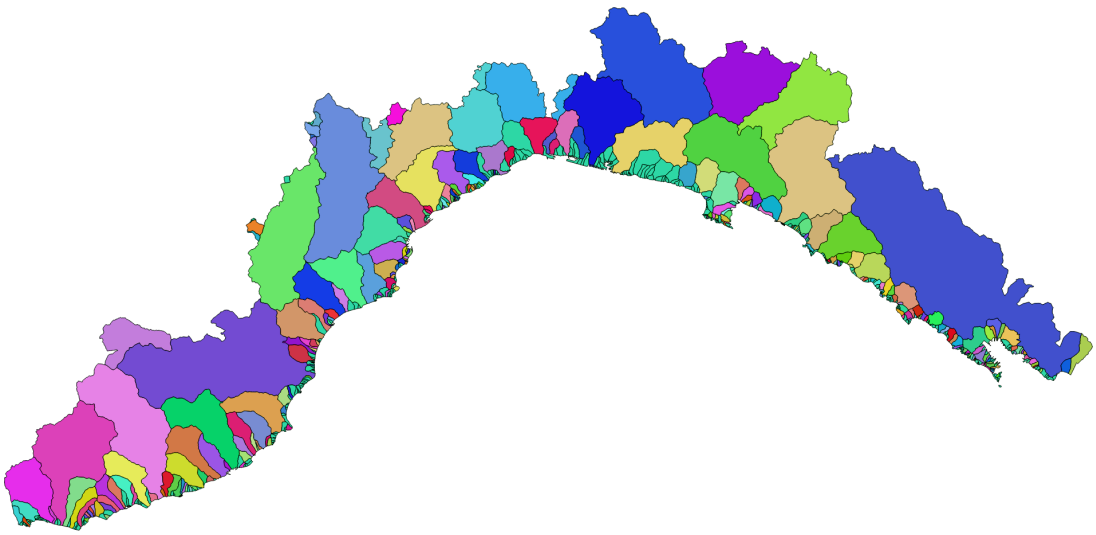


Figure 1.2 Map of drainage basins in the Liguria region (randomly coloured)

1.9 Relevant publications

This thesis is partly based on previous, peer-reviewed work. In this section we describe how papers and project deliverables contributed to this thesis and specifically point out the advances we made since their publication. We also list works that did not contribute directly to this thesis but deal with similar topics or give further details on specific points. The list of publications is sorted by relevance.

Krämer, M., & Senner, I. (2015). A Modular Software Architecture for Processing of Big Geospatial Data in the Cloud. *Computers & Graphics*, *49*, 69–81. https://doi.org/10.1016/j.cag.2015.02.005

In this journal paper we present a first version of our software architecture. The paper has contributed to Chapter 2, *Architecture* but the text has been significantly updated and extended. This thesis includes a more detailed and elaborate description of the architecture, the components and their interfaces. In addition, we give a broader overview of the state of the art and describe how our work relates to it. Finally, we present a comprehensive requirements analysis that was not part of the original work. Although the paper included a few results from an initial evaluation, Chapter 5, *Evaluation* is new and incorporates the advances we made since the publication of the paper.

Krämer, M., Skytt, V., Patane, G., Kießlich, N., Spagnuolo, M., & Michel, F. (2015). *IQmulus public project deliverable D2.3.2 - Architecture design - final version*.

This deliverable from the IQmulus project also describes an earlier version of our architecture. It contributed some technical details to Chapter 2, *Architecture*. The structure of the chapter is to a certain extent similar to the deliverable but the text has been significantly updated or rewritten. New sections have been added such as the comparison to the state of the art, the requirements analysis and the discussion on system operations.

Krämer, M. (2014). Controlling the Processing of Smart City Data in the Cloud with Domain-Specific Languages. In *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC)* (pp. 824–829). IEEE.

In this conference paper we present a modelling method for Domain-Specific Languages. Chapter 4, *Workflow Modelling* is partly based on this earlier work. The text has been updated and new sections, such as the application of the modelling method to our use case B, were added.

Krämer, M., & Senner, I. (2015). *IQmulus public project deliverable D2.4.2 - Processing DSL Specification - final version*.

This project deliverable describes a Domain-Specific Language that is comparable to the one we present in Chapter 4, *Workflow Modelling*. Our use cases are similar to the ones in the deliverable, but in this thesis we discuss related work in detail, we give a full overview over our grammar, and we describe our user interface (the workflow editor). In addition, we present a way to interpret workflow scripts written in the Domain-Specific Language and define how they can be mapped to executable actions.

Hiemenz, B., & Krämer, M. (2018). Dynamic Searchable Symmetric Encryption in Geospatial Cloud Storage. *International Journal of Information Security*. Submitted, under review.

In this journal paper we present a method to store geospatial data securely in the Cloud, based on Searchable Symmetric Encryption. It contributed to Section 2.7.4 on Cloud-based data storage and partly to Section 2.13 on security.

Krämer, M., & Frese, S. (2019). Implementing Secure Applications in Smart City Clouds Using Microservices. Submitted, under review.

This journal paper has been written in parallel with this thesis. It describes another software architecture based on microservices that enables secure Smart City applications in the Cloud. The paper has contributed to Section 2.1.2 on microservice architectures and partly to Section 2.2.1 on related work.

Böhm, J., Bredif, M., Gierlinger, T., Krämer, M., Lindenbergh, R., Liu, K., … Sirmacek, B. (2016). The IQmulus Urban Showcase: Automatic Tree Classification and Identification in Huge Mobile Mapping Point Clouds. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLI-B3*, 301–307. https://doi.org/10.5194/isprs-archives-XLI-B3-301-2016

This conference paper gives further details on our use case A, in particular in terms of the algorithms used to process the urban data and the visualisation of the results.

Krämer, M., Zulkowski, M., Plabst, S., & Kießlich, N. (2014). *IQmulus public project deliverable D3.2 - Control Components - vertical prototype release*.

This deliverable from the IQmulus project is worth noting because it gives an overview of the processing chain (from interpreting workflow scripts written in a Domain-Specific Language to executing them in the Cloud).

1.10 Structure of the thesis

The thesis is structured along the three pillars described in Section 1.6. We start with a detailed description of our software architecture in Chapter 2, *Architecture*. We include a comprehensive requirements analysis, interface descriptions, and a discussion on topics related to operations and security.

Chapter 3, *Processing* presents details on our component for workflow execution. We describe interfaces as well as the internal control flow in the individual parts of our component. The chapter also includes a definition of service metadata which enables developers to integrate their services into our architecture.

The third pillar is covered by Chapter 4, *Workflow Modelling* where we present our method for the modelling of Domain-Specific Languages as well as the language we use to describe workflows for our use cases. We also include a description of a user interface for workflow definition (a workflow editor) and describe how language elements can be mapped to executable actions.

In order to validate if our software architecture is suitable to execute workflows from real-world use cases, we present a comprehensive evaluation in Chapter 5, *Evaluation*. We perform a quantitative evaluation where we apply our system to our use cases, as well as a qualitative discussion on the requirements defined in earlier chapters and how our system satisfies them.

We finish the thesis with conclusions and a discussion on future research.

2 Architecture

In this chapter we present our architecture for the processing of large geospatial data in the Cloud. The main goal of our architecture is to provide both GIS users and developers of spatial processing algorithms with the means to leverage the capabilities of the Cloud. Our architecture is scalable and supports processing of arbitrarily large data sets. Its design is based on the microservice architectural style. One of the key points of our architecture is that it enables distributed development. Developers and researchers from different companies and institutions can contribute their processing algorithms and extend the functionality of our system. Due to the modularity of the architecture, such external components can be integrated without fundamental modifications. The architecture is also designed to be fault tolerant and highly available.

The chapter is structured as follows. We first provide the reader with background on Service-Oriented Architectures and the microservice architectural style. We then present existing work and describe how our architecture relates to it. After that, we perform a comprehensive requirements analysis by defining stakeholders that have an interest in our system as well as quality attributes our architecture should meet. The main part of the chapter describes the overall architecture, its components, and how they communicate with each other. We also define a few technical requirements that processing algorithms need to satisfy in order to be integrated into our system. Further, we discuss continuous deployment and operational aspects such as monitoring and logging. We finish the chapter with a summary.

2.1 Background

In this section we discuss two architectural styles of software design that are of major importance for our work: the Service-Oriented Architecture (Section 2.1.1) and the microservice architectural style (Section 2.1.2) which emerged from the former and provides the basis for our system.

2.1.1 Service-Oriented Architecture

Service-Oriented Architecture (SOA) describes a style of designing a distributed application with loosely coupled components (services) communicating over network protocols, in contrast to a

monolithic application where components are tightly coupled and communicate through function calls inside the same process space. The main goal of SOA is to provide means to create large distributed systems that are scalable and flexible. For lack of a common and concise definition for SOA, Footen & Faust (2008) have created the following one:

> SOA is an architecture of independent, wrapped services communicating via published interfaces over a common middleware layer.

Josuttis (2009) proposes employing an Enterprise Service Bus (ESB) as the middleware layer. An ESB decouples the services, provides a higher degree of interoperability, and reduces the number of communication channels. Instead of communicating directly with each other, the services only need to connect to the ESB. The bus handles network protocols and message routing, and supports multiple message exchange patterns (e.g. asynchronous request/response or publish/subscribe).

Figure 2.1 depicts a Service-Oriented Architecture with five services connected through a middleware layer. The diagram also shows how an existing software component (often called a *legacy service*) can be integrated into an SOA by providing a wrapper service that handles the communication with the middleware layer on behalf of them. This pattern allows a Service-Oriented Architecture to be implemented in a company incrementally without the need to completely rebuild the company's infrastructure from scratch.

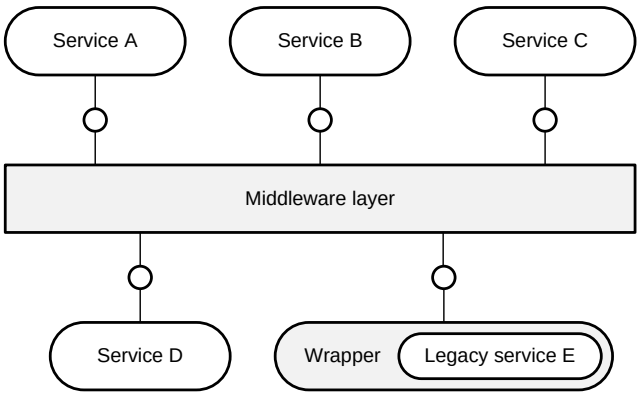


Figure 2.1 A Service-Oriented Architecture according to Footen & Faust (2008)

The term *Service-Oriented Architecture* was originally coined by Gartner (Schulte, 1996; Schulte & Natis, 1996). It gained momentum in the early years of the 21st century with the boom of the Internet and the World Wide Web, which became available to a broad audience. New web technologies and network protocols made it easier to create an application of distributed loosely coupled services. Major drivers were technologies such as HTTP, XML, and SOAP. Large companies such as IBM, Oracle, HP, SAP and Sun joined the momentum and created a whole ecosystem around SOA consisting not only of tools, technologies, and design patterns on the technical level, but also extending to the business level where common enterprise roles, policies and processes were defined. This created criticism by people who considered SOA just a hype and a buzzword with which IT vendors tried to make money by selling concepts and tools or simply rebranding old ones (cf. Josuttis, 2009).

Due to the fact that SOA and the World Wide Web experienced a boom almost at the same time, a Service-Oriented Architecture was (and still is) often considered equivalent to a distributed web application consisting of web services. However, Natis (2003) states the following:

> […] Web services do not necessarily translate to SOA, and not all SOA is based on Web services […]

SOA should rather be seen as the continuation of object-oriented programming (OOP) on a higher level. Much like OOP is used to modularise programs, SOA can be used to split a large application into a set of distributed services, each of them having their own responsibilities. Whether these services use web technologies or not is actually irrelevant. According to the Open Group's definition of SOA (Footen & Faust, 2008, p. 72), a service is a component that has the following properties:

- It is self-contained
- It may be composed of other services
- It is a black box to consumers of the service

Most of the services we describe in our work—in particular the processing services (see Section 2.6)—are not web services but still have these properties.

The fact that people confused SOA with web services, as well as the criticism around exploiting the term commercially, led to a constant decline of popularity. In addition, the policies and business processes specified and promoted by large IT vendors often did not match the structures of other organisations. On a technical level, SOA imposed a couple of limitations. An enterprise service bus does not always fit in any distributed application. Technologies such as XML and SOAP were considered too heavy, too complex, and out of date compared to their more modern and lightweight counterparts JSON and REST.

With the advent of Cloud Computing a more flexible way to create large distributed systems was required. This led to the creation of the microservice architectural style.

2.1.2 Microservice architectural style

The term *microservice* is not clearly defined in the literature yet. The British software engineer Martin Fowler (2014) describes it as follows:

> In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

According to this, an application based on microservices is—as opposed to a large monolith—split into small pieces acting in concert to serve a larger purpose. Each of these pieces is developed, maintained, and deployed independently. Microservices have the following characteristics:

Size and focus. Each microservice typically serves one specific purpose. For example, a distributed system used in a supply company may contain a service for customer management, one for stock management, and another one for the processing of orders. The boundaries between microservices are often drawn along so-called *Bounded Contexts* which are identified while specifying the application's architecture using *Domain-driven Design* as described by Evans (2003).

Independence. Microservices are *separated* from each other or *autonomous* (Newman, 2015). They run in their own processes—often even in separate virtual machines or containers. They offer interfaces to other services and communicate through lightweight protocols such as an HTTP API as described above. The fact that they run in their own processes also means they are separated programs and projects. They are developed independently and often by different people or teams. These teams use the technologies and programming languages they are familiar with but not necessarily the same as other teams working on the same distributed application. One of the biggest advantages of this separation is the fact that microservices can be deployed to production independently. This means new features and bug fixes can be made available to customers in a short amount of time and without affecting the overall availability of the distributed application.

Scalability and fault tolerance. Modern distributed applications need to be *scalable* to be able to handle large amounts of users and data. They also need to be *resilient* to external influences such as a quickly growing number of customers (e.g. on busy shopping days such as the Black Friday in the U.S. or before Christmas), as well as failing components (e.g. broken hardware, unstable network connection or crashed software components). Microservices can help implement a scalable and resilient system. They are deployed in a distributed manner and typically redundantly. Peaks in demand can be handled by adding more service instances. If one of them should fail or become unavailable—for whatever reason—other instances can take over. In any case, even if all instances of a microservice should fail, the impact on the rest of the application is minimized.

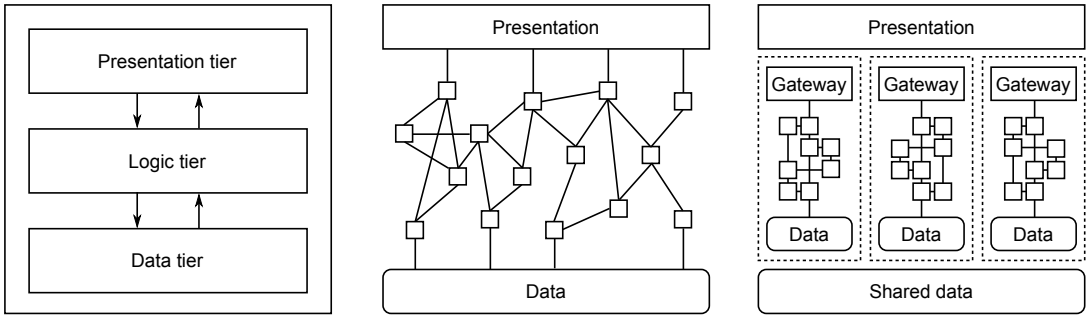
Organisational patterns. According to *Conway's Law*, "organizations which design systems […] are constrained to produce designs which are copies of the communication structures of these organizations" (Conway, 1968). This means the architecture of any software mirrors the structure of the organization developing it. For example, if you assign five teams to develop a distributed application you will most likely get an architecture consisting of five different services communicating with each other. Microservices match this concept very well. As described above, they are developed independently by different teams. Each team is responsible for one or more microservices, but one service never falls into the responsibility of more than one team. In projects where many parties provide services, this approach helps keep responsibilities clear and enables distributed collaboration.

Composability. Microservices are *composable* and *replaceable*. Multiple microservices act in concert and make up a larger application. Single services may be reused in different applications or in different areas of the same application. In addition, since microservices are small and serve only one purpose they can be easily replaced, for example, if they are updated to a new technology stack, or if a newer service version provides an improved algorithm or better security.

The microservice architectural style is quite similar to the approach of a *Service-Oriented Architecture*. There are subtle differences, particularly in terms of the actual implementation of a distributed application and guidelines for how to design the architecture. Sam Newman (2015, p. 9, first paragraph) summarises the differences as follows:

> The microservice approach has emerged from the real-world use, taking our better understanding of systems and architecture to do SOA well. So you should instead think of microservices as a specific approach for SOA in the same way that XP or Scrum are specific approaches for Agile software development.

A microservice architecture is therefore a Service-oriented Architecture. The difference is most apparent in the way services are deployed and executed. While SOA defines that an application should consist of services, it does not define how they should be run. In an SOA services may still be part of a single monolithic application (often running inside an application container such



a) A monolithic software with a standard three-tier architecture

b) A pure microservice architecture with a complex deployment

c) A microservice architecture decomposed along bounded contexts

Figure 2.2 Comparing a monolithic software architecture with two approaches to system decomposition based on microservices

as Apache Tomcat) whereas the microservice architectural style demands that these services must run in separate processes and be deployed independently. The microservice approach therefore goes one step further and gives guidance and concrete instructions in areas where SOA is lacking clarity (Newman, 2015).

In a microservice architecture a middleware layer does not play such an important role as in an SOA. If necessary, an ESB can still be used—but it is not required—which allows developers to create more flexible and arbitrary networks of services. The microservice architectural style also does not specify policies and business processes that enterprises must follow. It is therefore a more lightweight approach than a full-fledged SOA and mostly applies at the technical level without impinging on the way companies conduct their business.

While the microservice approach has many advantages, it also comes with a couple of drawbacks. For example, the application's complexity increases with the growing number of services. This problem is known from monolithic applications whose complexity increases with the number of classes or components, but is further aggravated by the fact that the services are distributed and have to communicate over a network that may be unreliable (see Deutsch, 1994).

The complexity can be tackled by decomposing an application vertically according to bounded contexts. Figure 2.2 compares a typical monolithic application to a software architecture based on microservices. The standard way to implement a monolithic application is to divide it into three tiers, namely the data tier, the logic tier and the presentation tier (Figure 2.2a). An application based on Microservices is a set of loosely coupled services communicating with each other (Figure 2.2b). The larger the application becomes, the more important it will be to order the services. Figure 2.2c shows how an application can be decomposed vertically along bounded contexts. That means that services belonging to the same context (such as those managing customers, stock and orders, as well as those dealing with employees and internal accounting) should be grouped and only a couple of services are allowed to communicate with services from other bounded contexts.

For our work, the microservice architecture style is beneficial as it allows us to create a system that is very flexible, scalable and fault-tolerant. As we will show in Section 2.3.1, services in our architecture can be developed by teams distributed over multiple countries. The microservice architectural style allows these teams to work independently and yet integrate their services into a single system. Furthermore, microservices can be reused and composed in different ways in order to satisfy varying requirements. This property is essential for our work, because it allows us to orchestrate services to complex spatial processing workflows.

2.2 Related work

After we discussed relevant architectural styles for software design in the previous section, we now review the literature relevant to our work. We first show how microservice architectures are used in the scientific community to build various systems. Then we discuss software architectures for distributed data processing as well as geospatial applications in the Cloud. Finally, we provide an overview of relevant work combining these two areas, namely the processing of geospatial data in the Cloud.

2.2.1 Microservice architectures

Since the advent of microservices and their broader acceptance within the industry and also the scientific community, the number of publications on this topic has increased quickly. One of the most influential work on microservices is the book "Building microservices" by Newman (2015). The book touches many aspects related to microservices ranging from how to model services and how to integrate them into larger systems, up to operational topics such as deployment and monitoring.

The microservice architectural style has significant benefits over the traditional way of designing an application as a monolith. Villamizar et al. (2015) compare both approaches and specifically focus on applications in the Cloud. They argue that monolithic applications are often not designed to be run in the Cloud and therefore cannot handle dynamic infrastructure changes. A microservice architecture, on the other hand, is more flexible and scalable as individual services can be deployed and relocated on demand during runtime. Villamizar et al. state that there are a number of factors that can increase the complexity of a system: the number of services, the number of involved developers and teams, the number of operators, and the number of targeted business applications. They claim that the microservice architectural style can help tackle the complexity. As we will describe in Section 2.3.1, we have similar issues in our architecture, in particular since we have a large number of services (more than a hundred, see Chapter 5, *Evaluation*) and many distributed development teams.

Villamizar et al. also describe challenges linked to the microservice architectural style. The development of a system of distributed services can sometimes be hard because issues such as network failures or timeouts need to be considered. In addition, team management processes should be adapted to this new style of system development. This is, however, not just a drawback but can also be a chance to increase efficiency in development and collaboration between developers and teams. Balalaie, Heydarnoori, & Jamshidi (2016), for example, report on their experience with applying the microservice architectural style and methods from the DevOps movement (Loukides, 2012) to the development of an application providing data management functionality to mobile developers. They formed small cross-functional teams—each of them being responsible for one service—as well as a core team for shared functionality. They state that one of the main benefits of this approach is a shorter time-to-market. The teams are completely responsible for their service and can deploy it much faster than a traditional team structure consisting of development, quality assurance and operations could. In addition, Balalaie et al. further state that code written by such small teams has a higher comprehensibility and maintainability, and that new team members can be added with a lower learning curve.

One of the key findings from the work of Balalaie et al. is that automated deployment plays an important role in a microservice architecture, in particular with a growing number of services and a higher complexity of the infrastructure the services are deployed to. A similar observation is made

by Ciuffoletti (2015) who describes a microservice-based monitoring infrastructure and how it can be automatically deployed. He argues that automated deployment can also help transfer an application to different infrastructures for development, testing and operation (portability). In our work we also automate the deployment of our system in order to be able to handle its complexity and the large number of services. Furthermore, we consider portability, in particular to keep the system independent from the infrastructure it is installed on and thus avoid vendor lock-in.

In order to implement the automated deployment, Ciuffoletti makes use of container technology. Containers have major benefits in terms of isolation and portability of services. However, containerisation is a kind of virtualisation and can affect performance. Independent studies conducted by Amaral et al. (2015) and Kratzke (2015) show that while the impact on CPU power is negligible, network performance inside containers is significantly slower than on the hosting machine (up to 20%). In our architecture we also use container technology to deploy services. Nevertheless, we do not transfer large amounts of data between processing services through the network. Instead, we use a distributed file system as the main communication channel and mount it into the containers when we start them. The communication between compute nodes in the Cloud is managed by the driver of the distributed file system which is not containerised. Since the performance impact on file system I/O operations inside containers is very small (Felter, Ferreira, Rajamony, & Rubio, 2015) our processing services run almost as fast as if they were not containerised.

One major challenge in applying the microservice architectural style is that with a growing number of services a distributed application can become very complex which may lead to security vulnerabilities. Esposito, Castiglione, & Choo (2016) state that having multiple services can enlarge the attack surface by offering more vulnerability points. They also argue that the microservice architectural style encourages using off-the-shelf software (such as open-source libraries) and that their trustworthiness should be properly validated and monitored. We discuss some security aspects in Section 2.13 but a comprehensive security concept is beyond the scope of this work.

Running a microservice architecture in the Cloud is beneficial to many companies, in particular considering the costs in comparison to operating a monolithic application. Villamizar et al. (2016) report on a case study they conducted to compare a monolithic architecture to one based on microservices and another one running serverless on Amazon AWS Lambda. They implemented an example application using these three different architectures and compared performance and response times but particularly focused on the costs. They conclude that microservices can help reduce infrastructure costs tremendously but the increased effort of implementing and maintaining an application based on this architectural style has to be considered carefully.

An approach to manage the effort is presented by Toffetti, Brunner, Blöchlinger, Dudouet, & Edmonds (2015). They propose a microservice architecture that enables scalable and resilient self-management of Cloud applications. They employ distributed storage and leader election functionalities of existing tools commonly used in Cloud application development. Their approach allows them to constantly monitor their application and to implement features such as autonomous health management and self-healing. One of the main contributions of their work is that they do not rely on infrastructure provider services and therefore avoid vendor lock-in. They conclude that the microservice architectural style aligns well with their approach.

There are a couple of papers dealing with the experiences from applying microservices to practical use cases. For example, Vianden, Lichter, & Steffens (2014) present a reference architecture for Enterprise Measurement Infrastructures (EMIs) as well as two case studies in which they apply this architecture to an EMI monitoring software development and another one collecting risk metrics in IT projects. They argue that systems based on classic SOA suffer from centralized integration problems such as the need for a common data schema and related mapping issues. To avoid these problems, they divide their EMI into dedicated microservices for measurement, calculation and visualization. Their results look promising and they suggest further long-term field studies.

Alpers, Becker, Oberweis, & Schuster (2015) present a microservice-based architecture to create tool support for business process modelling and analysis. They claim that the microservice architectural style improved the scalability of their system, and that additional or different functionality could be easily implemented.

The microservice architectural style has also been applied to data-driven workflow management. Safina, Mazzara, Montesi, & Rivera (2016) present Jolie, a programming language to formalise the composition of microservices to create data-driven workflows. They claim that their approach helps identify common communication patterns in microservice architectures, which opens opportunities for new programming scenarios.

For supplemental information on the state of the art and publications related to microservices we refer to the work by Alshuqayran, Ali, & Evans (2016) who present a systematic study on the research conducted in this field including architectural challenges faced by the community, diagrams used to represent microservice architectures, as well as involved quality requirements. In addition, Di Francesco, Malavolta, & Lago (2017) present results from a thorough review of the literature on the microservice architectural style. They specifically evaluate existing work from three perspectives: publication trends, focus of research, and potential for industrial adoption. They state that most publications on microservices are of a practical nature and that they present specific solutions. They also claim that the research field is still immature and that most of the studies they reviewed are far away from being transferred to industrial use. However, they note that the "balanced involvement of industrial and academic authors is […] promising for knowledge co-creation and cross-fertilization".

2.2.2 Architectures for Big Data processing

There are a couple of architectural patterns that are often used for Big Data processing applications. Each of these patterns targets specific use cases and has different benefits and drawbacks. In this section we review the most prominent approaches and compare them to ours.

Batch processing

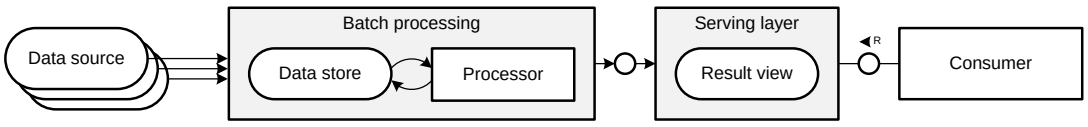


Figure 2.3 A batch processing architecture

Batch processing works well for applications where data is first acquired and subsequently processed. The processing pipeline is depicted in Figure 2.3. The data is collected from one or more data sources and then put into a data store. The processing can be triggered any time and operates on the whole set of collected data or on a smaller batch of it. It can potentially happen in an iterative way. Intermediate results are written back into the store, but the original data is never changed. Final results are sent to a serving layer which produces a result view for consumers to query.

A typical programming pattern for batch processing is MapReduce (Dean & Ghemawat, 2008) which makes batch processing very scalable. It can handle arbitrary data volumes and can be scaled out by adding more resources—typically compute nodes. The most popular frameworks for batch

processing are Hadoop and Spark. Tools such as HBase, Imapala, or Spark SQL can be used to implement the serving layer and to provide interactive queries on the result view.

Stream processing

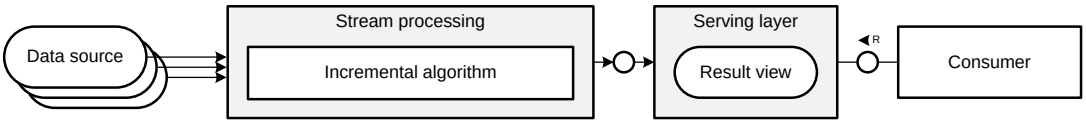


Figure 2.4 A real-time stream processing architecture

One drawback of batch processing is that it can take a long time (a couple of hours or even longer) if the input data set is very large. Applications that have to provide information in near real-time need a faster approach. In a stream processing system as depicted in Figure 2.4, incoming data is handled immediately. A stream-oriented processing pipeline is event-driven and has a very low latency. Results are typically produced in the order of less than a second. In order to achieve this, the result view in the serving layer is updated incrementally.

Immediately processing each and every single event can introduce overhead. Some stream systems therefore implement a micro-batching approach where events are collected to very small batches that can still be processed in near real-time.

Typical frameworks for stream processing are Spark Streaming, Storm, or Samza. Data storage solutions that support incremental updates and interactive queries are Cassandra and Elasticsearch.

Lambda architecture

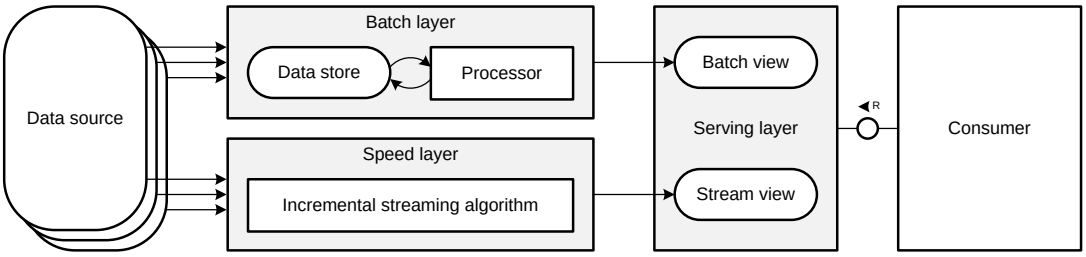


Figure 2.5 The Lambda architecture

While stream processing can provide results in a short time, it is not very resilient to changes in the processing algorithm code. Such changes can happen if there was a bug in the code or if requirements have changed and additional values need to be computed from the input data set. Batch processing allows the result view to be recomputed by processing the whole data set again. In the stream processing approach, on the other hand, there is no store for input data and hence recomputing is not possible. A bug in the processing code can corrupt the incremental result view without a way to make it consistent again.

In order to combine the benefits of both approaches—the fault-tolerance of batch processing and the speed of stream processing—Nathan Marz has created the Lambda architecture for Big

Data processing (Marz & Warren, 2015). In this architecture, input data is sent to a batch layer and a so-called speed layer at the same time (see Figure 2.5). Both layers implement the same processing logic. The batch layer is used to process large amounts of data and to regularly reprocess the whole input data store. The speed layer contains a stream processing system and compensates for the high latency of batch processing. It only calculates results for data that has arrived since the last run of the batch layer.

In the serving layer, the batch results as well as the incremental streaming results are combined to a view that provides a good balance between being up-to-date and correct (i.e. tolerant to errors). Streaming results are discarded as soon as new batch results have been computed.

Kappa architecture

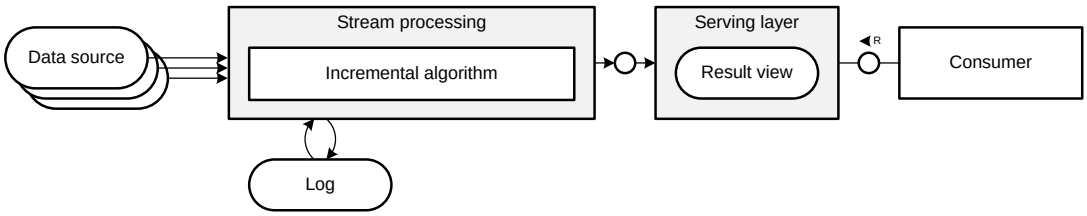


Figure 2.6 The Kappa architecture

Modern stream processing systems are as capable as batch processing systems in terms of functionality and expression power. If they were not, it would actually not be possible to implement a Lambda architecture, where both branches have the same processing logic. Due to this, people have started questioning the usefulness of batch processing in the Lambda architecture. Maintaining two branches with the same logic in two different programming styles can become very complex. The only advantage of batch processing over stream processing is its resilience to programming errors or changing requirements, which is based on the fact that the original input data is permanently kept in a store and recomputing is therefore always possible.

In an attempt to simplify the Lambda architecture, Kreps (2014) has created the Kappa architecture (see Figure 2.6). This architecture is very similar to a typical stream processing pipeline. However, Kreps recommends keeping a log of incoming data (or events) which can be replayed and therefore used to recompute the result view if necessary. A typical framework that allows for collecting input data in a log for a certain amount of time (retention period) is Kafka. In order to cover most cases, it is recommended to configure a long retention period of at least several weeks. Since Kafka allows for processing the log from any point and by multiple readers at the same time, recomputing the result view can be done without interrupting near real-time processing.

Summary

In this section we have described four of the most common architectures for Big Data processing. The one that is best comparable to ours is batch processing. Although velocity—i.e. the speed in which data is acquired and has to be processed—often plays an important role for geospatial applications, data acquisition and processing are typically separate events with defined start and end points (see Kitchin & McArdle, 2016). For example, in our urban planning use case described in Section 1.8.1 mobile mapping data is first collected on a hard disk and then uploaded to the

Cloud for processing. A similar pattern can be found in our second use case (see Section 1.8.2) or in applications dealing with satellite imagery.

In contrast to information collected by a social media platform, for example, the acquisition of geospatial data is usually not a continuous process but one that is inherently eventually finished —i.e. as soon as all required data has been collected. The stream processing approach expects data to be collected continuously and aims at processing it at the same time with a short latency. In order for this to work and to meet near real-time constraints in the order of less than a second, the individual processing steps need to be very fast. Geospatial algorithms and the models they operate on are, however, known to be very complex and expected to become even more so in the future (Yang et al., 2011). Near real-time is possible for certain use cases such as the evaluation of readings in a sensor network but not reasonable for every geospatial application.

The architectures presented in this section are typically used to implement very specific use cases and processing pipelines. Our architecture, on the other hand, allows for creating more flexible workflows that can be used for a wider range of purposes. In fact, as we will show later, we can incorporate batch or stream processing into our workflows and thus create pipelines on a much higher level.

2.2.3 Cloud architectures for geospatial applications

One of the major research topics in the geospatial community in recent years is modern urban management and the use of supporting information technology. A city that leverages ICT to improve urban development processes is often referred to as a *Smart City* (Ludlow & Khan, 2012). To achieve their goal of a sustainable and liveable urban environment, Smart Cities collect and process large amounts of geospatial data. Cloud Computing technology is often used in this context (Khan, Anjum, & Kiani, 2013).

A major amount of urban data is collected by IoT devices (Internet of Things) distributed all over the city (stationary as well as mobile devices). For example, in-situ sensors produce data about traffic density, air quality or weather that can be used to analyse events in near real-time and to produce plans to counteract negative implications of, for example, recurring traffic jams and their impact on the environment. At the same time, data from mobile devices such as smartphones, tablets or sensors in vehicles can be used to analyse motion patterns of citizens and monitor socio-economic interactions and developments, which helps implement measures eventually leading to a higher quality of life (Peters-Anders, Loibl, Züger, Khan, & Ludlow, 2014).

Krylovskiy, Jahn, & Patti (2015) argue that IoT is one of the key enablers of Smart City applications. They present a software platform which aims at engaging various stakeholders in order to increase the energy efficiency of urban districts. They identify a number of challenges that developers are typically faced with when designing a scalable IoT platform: a large variety of services, constantly evolving technologies, changing requirements, interdisciplinary and international teams, as well as the demand to increase quality while reducing operational costs. They state that there are similar challenges in the development of distributed applications and that the microservice architectural style simplifies the design and implementation of individual services. They also argue that it comes with the cost of a more complex distributed system in which compromises such as eventual consistency are, however, reasonable trade-offs for the gained benefits.

Related publications on Smart City applications in the Cloud often do not specifically focus on data processing and computational power, but also try to leverage other capabilities such as distributed data storage and the possibility to make information accessible at a central location in order to share it with third parties—e.g. other departments or official agencies in the same municipality as well as the public. Khan & Kiani (2012), for example, present an architecture for context-aware citizen services. The main purposes of this architecture are data collection and

centralised access. Khan & Kiani do not use microservices. They divide their system into seven distinct layers ranging from platform integration, data acquisition and analysis, to the actual application. They conclude that the processing and data storage capabilities of Cloud Computing provide a suitable environment for Smart City applications. One of the major benefits of their approach is that information from different sources can be integrated at a central location and enriched with contextual data in order to enhance the experience of citizens with their system.

In parallel with this thesis, we have also explored the use of microservices for Smart City applications, but we specifically focused on secure data storage (Krämer & Frese, 2019). We created a distributed application for the assessment of security risks in urban areas. Since such an application has to deal with potentially sensitive data, we presented an approach leveraging attribute-based encryption in order to store large data sets securely in the Cloud, while preserving the possibility to share them with multiple stakeholders. The microservice architectural style helped us to create a scalable and flexible design that can be applied to other use cases too.

Cloud Computing can also be beneficial for geospatial applications in general, not only for Smart Cities. Lee & Percivall (2008) state that the "ability to access, integrate, analyse, and present geospatial data across a distributed computing environment […] has tremendous value" but also requires standardised interfaces. They argue that international standards for distributed geospatial applications are required in order to improve interoperability between systems and to ease data access. The OGC (Open Geospatial Consortium) has recently set up a new domain working group for Big Data dealing with Service-Oriented Architectures for the distributed processing of spatial data and the standardisation of related technologies. The architecture we present in this chapter is very flexible and allows for integrating various kinds of processing services. This also includes web processing services such as the OGC WPS. The OGC is of major importance for the geospatial community and we consider the possibility of integrating OGC services into our system an advantage. However, OGC services are web-based and have an HTTP interface. This means data has to be transferred through HTTP before it can be processed which may impose a certain performance hit. In our architecture we deploy a distributed file system (see Section 2.7.1) to which processing services are directly connected. This allows for a faster data access. Nevertheless, including services such as the OGC WMS (Web Map Service) or WFS (Web Feature Service) as an external data sources can be beneficial if it improves the quality of processing results.

2.2.4 Cloud architectures for geospatial processing

While there has been work on Big Data processing as well as on Cloud architectures for geospatial applications, the combination of the two, Cloud architectures for geospatial data processing, has only become subject to research in the last couple of years (Agarwal & Prasad, 2012; Cossu, Di Giulio, Brito, & Petcu, 2013). The availability of commercial Cloud solutions such as Amazon Web Services (AWS) or Microsoft Azure has facilitated applications in this area. For example, Qazi, Smyth, & McCarthy (2013) describe a software architecture for the modelling of domestic wastewater treatment solutions in Ireland. Their solution is based on AWS on which they install the commercial tool ArcGIS Server via special Amazon Machine Images (AMIs) provided by Esri. Qazi et al. make use of the ArcGIS Server REST interface to deploy web services providing spatial datasets. Additionally, they implement a web application that can be used for decision support. While the focus of their work is on deploying a highly available data storage and a decision support tool, they do not cover the issue of very large geospatial datasets and how the capabilities of Cloud Computing can be exploited to process them. In addition, their work depends on the commercial ArcGIS Server and the respective Amazon Machine Images. Our architecture, on the other hand, does not rely on commercial software and can be configured to run on different infrastructures.

Warren et al. (2015) report from their experience from processing over a petabyte of data from 2.8 quadrillion pixels acquired by the US Landsat and MODIS programs over the past 40 years. They claim to be the first researchers who are able to process such a large data set in less than a day. They leverage the Cloud Computing platform from Google. Their processing pipeline consists of 10 steps including uncompressing raw image data, classifying points, cutting tiles, performing coordinate transformations, and storing the results to the Google Cloud Storage. The process is static and highly optimised for the Google platform. The architecture we present, on the other hand, is portable and allows more configurable and parametrisable workflows to be created.

Li et al. (2010) leverage the Microsoft Azure infrastructure to process high-volume datasets of satellite imagery in a short amount of time. Their solution consists of a cluster of 150 virtual machine instances which they claim to be almost 90 times faster than a conventional application on a high-end desktop machine. They achieve this performance gain by implementing an algorithm based on reprojecting and reducing. This approach can be compared to MapReduce. However, it was explicitly developed for the Azure API which provides a queue-based task model quite different to MapReduce. Compared to their approach, our work does not focus on one specific processing model. Instead, we describe an architecture that is flexible and facilitates a number of different approaches to distributed algorithm design.

Since a growing number of Cloud infrastructure providers support MapReduce—in particular its open-source implementation Apache Hadoop—the geospatial community has started developing solutions specifically targeted at this. The Esri GIS Tools for Hadoop, for example, include a number of libraries that allow Big Geo Data to be analysed in the Cloud. The libraries are released as open source. They offer a wide range of functionality including analytical functions, geometry types and operations based on the Esri Geometry API. Similar to this, the project SpatialHadoop adds spatial indexes and geometrical data types and operations to Hadoop. While there has been work utilising the Esri GIS Tools for Hadoop and SpatialHadoop (Ajiy et al., 2013; Eldawy & Mokbel, 2013), the MapReduce paradigm implies fundamental changes to geospatial algorithm design as it has been done before. The effort of migrating an existing algorithm to MapReduce often outweighs its advantages. MapReduce is not the only solution to exploit Cloud Computing infrastructures. Other approaches such as actor-based programming or in-memory computing are often more appropriate for certain algorithms and in some cases a lot faster (Xin et al., 2013). Our architecture enables arbitrary algorithms to be executed in the Cloud, which allows developers to select the most appropriate programming paradigm for a specific purpose.

One of the most popular frameworks for distributed data processing that goes beyond MapReduce is Apache Spark. Liu, Boehm, & Alis (2016) use this framework to detect changes in sets of large LiDAR point clouds. They conclude that Spark is suitable to process data that exceeds the capacities of typical GIS workstations. However, they are not completely satisfied with the results of their processing algorithm and propose adding a postprocessing step to reduce the noise. Solutions such as Hadoop or Spark are suitable to create specific processing algorithms but for workflows where a chain of algorithms is applied (e.g. preprocessing, change detection, and postprocessing) a higher-level solution is necessary. Our architecture allows such workflows to be created. It can control simple services but also processing frameworks such as Hadoop or Spark and integrate them into a workflow.

Since geospatial applications in the Cloud are quite new, the community is still looking for best practices. Agarwal & Prasad (2012) report from their experience with implementing a Cloud-based system called Crayons that facilitates high-performance spatial processing on the Microsoft Azure infrastructure. They present several lessons learnt ranging from data storage to system design. In particular, they state that a large system should be designed with an open architecture so individual components can be replaced by others without affecting the overall system functionality. Our architecture is service-oriented and consists of loosely coupled components that can be exchanged quite easily. This way, a wide range of spatial processing services are supported and

can be extended later without requiring changes to the architecture. Additionally, our approach allows individual components to be replaced if requirements should change in the future.

2.3 Requirements analysis

In this section we describe the requirements that led to our software architecture. We first provide a list of stakeholders and what specific requirements and concerns they wish the system to meet and guarantee. We then describe a set of quality attributes—i.e. properties that the system has to have in order to satisfy the needs of the stakeholders. The requirements and quality attributes were derived from analysing the problem domain in Section 1.3, from our work in various international research projects, as well as our experience from developing GIS products and collaborating with domain users from municipalities, regional authorities, federal agencies, and the industry over the last nine years.

2.3.1 Stakeholders

In the following we describe various stakeholders who have an interest in a system for the processing of geospatial data in the Cloud. These people have different responsibilities ranging from software development to infrastructure management. We also include stakeholders who use our system to carry out GIS projects, as well as business professionals who have an interest in exploiting the capabilities of our system or the data processed with it.

Note that the stakeholders we present here are not necessarily individual people but roles. Multiple people can have the same role, and a single person can have multiple roles. For example, *members of the system development team* are often also *integrators* and *testers*. Additionally, since the DevOps movement (Loukides, 2012) has become more and more prominent in recent years, the boundaries between software development and IT operations have blurred and *developers* are now often responsible for *deployment* and *administration* too.

We divide people who have an interest in our architecture into roles to get a clearer picture of the domain and to identify individual requirements. However, in our experience component developers need to be involved in integration, deployment and operations in order to better understand how their components need to be designed to work correctly in a distributed environment. This particularly applies to the role of *processing algorithm developers* who often do not have a background in computer science or programming of distributed applications.

Users (GIS experts)

Our system design targets users who are GIS experts. They have diverse backgrounds (i.e. in geography, surveying, geoinformatics, hydrology, etc.) and work for different organisations and authorities such as municipalities, regional authorities, national mapping agencies but also companies dealing with or contributing to geospatial projects. These users work with large datasets that they need to store and process. However, their local workstations often lack hard disk space, main memory, or computational power, and so the storage and processing of large geospatial data is at least challenging or merely impossible.

In addition, the GIS experts often need to share data with their colleagues, with other departments, as well as with other authorities and organisations. This applies to original datasets and

processing results. At the same time, they need access to external datasets to combine them with their own data.

In order to perform these tasks, the GIS experts want to harness the capabilities of the Cloud. They need an infrastructure that has enough resources to store large datasets and an interface to share them with other parties. They also want to use the computational power of the Cloud to save time in data processing.

The experts typically use a desktop GIS (e.g. ArcGIS or QGIS) which offers a number of spatial operations and processing algorithms. Most desktop GISs also provide a way to create automated processing workflows. These workflows specify which spatial operations should be applied to a dataset in order to produce a certain result. In order to perform the same tasks as in their desktop GIS, the experts need similar operations in the Cloud. They also need a way to run automated processing workflows in the Cloud.

The workflows in a desktop GIS are typically programmed in a general-purpose language (GPL) such as Python. Most GIS experts, however, do not have a background in computer science or workflow programming. In our architecture we use a Domain-Specific Language (DSL) which is tailored to the processing of geospatial data. Such a language is easier to learn than a GPL and enables GIS experts to define their own workflows without a deep knowledge of programming. Our DSL abstracts from the details of Cloud Computing and distributed programming so that the GIS experts can focus on the workflow instead of technical details.

Our system also offers a way to share workflow definitions with other users. This enables GIS experts to use pre-defined workflows in which they just need to set the location of the input datasets and modify the parameters of the processing algorithms. The DSL helps the experts to quickly understand a pre-defined workflow and to decide whether it is suitable for the envisaged task.

Users (Data providers)

The users of our system typically obtain geospatial datasets from companies specialising in data acquisition and preprocessing. These data providers use methods such as terrestrial or airborne laser scanning and photogrammetry to collect large amounts of raw input data. In order to produce high-quality datasets that can be used in practical applications, the input data needs to be processed and finally be converted to standardised file formats.

For this purpose, data providers require an infrastructure that allows them to specify automated processing workflows for large data sets. They have similar requirements as the GIS experts but use different spatial operations or algorithms. They also operate on much larger datasets that are often produced in a short amount of time and that need to be processed and made available to customers quickly.

The people working at data provider companies are often also GIS experts and have a similar background. They also want to use existing pre-defined workflows for automated tasks in order to save time. However, since they need to deal with varying conditions (due to new acquisition methods and hardware) as well as changing requirements from their customers, they also need to be able to adapt processing workflows. Our Domain-Specific Language helps these people to understand pre-defined workflows and to write their own if required.

Members of the system development team

The system development team consists of software analysts, designers, architects and developers. Their main responsibility is system development and implementation. They aim to create a system

with a maintainable and extensible structure and code base. They wish to assign clear responsibilities to components (*high cohesion*) and to keep the number of dependencies between components low (*loose coupling*).

The microservice architectural style helps the members of the system development team to implement components (i.e. services) that serve a specific purpose and that have a well-defined and stable interface to communicate with other components. This allows them to easily add new features to the system and to deliver them quickly to customers. With the smaller code-base of the individual microservices they can also fix bugs faster.

Developers of spatial processing algorithms

Besides the system development team that is responsible for the processing platform—i.e. the user interface and the components dealing with workflow and data management—there are other software developers who contribute individual spatial processing algorithms. These people are often experts in mathematics, physics, photogrammetry, geomatics, geoinformatics, or related sciences, but have no background in computer science and hence limited knowledge of programming of distributed applications. They developed algorithms in the past for other projects and now want to leverage the Cloud to process large geospatial datasets. To this end, they wish to integrate their algorithms in our architecture—i.e. to register them as *processing services* (see Section 2.6).

The services have very diverse interfaces and are implemented using various programming languages and for different platforms. Since the services are used in multiple projects, the developers seek a way to easily integrate them into our system without modification. Our lightweight service metadata specification (see Section 3.6.2) enables them to do so.

Most of the services are single-threaded and do not harness the capabilities of multi-core systems or distributed computing. Our JobManager can distribute these services to multiple compute nodes in the Cloud. This allows the service developers to parallelise their algorithms without changing their code.

The developers typically work for different companies and institutions that are distributed across many countries. The microservice architectural style enables distributed teams to work on multiple components at the same time and to contribute them to a single integrated system.

Integrators

Integrators are people who take the spatial processing algorithms as well as the different software components created by the system development team and integrate them according to our architecture. To this end, they require the following:

- A repository of software components to integrate (artefacts). The repository should be able to store metadata for each artefact including a unique component identifier, a version number, and information about how to contact the developers. Ideally, the repository should also offer a versioning system so that the integrators can always access all versions of each service.

- Components with well-defined interfaces that can be integrated without manually creating additional middleware components, converters, or wrappers.

- Well-defined interface descriptions for processing services with arbitrary interfaces.

The microservice architectural style has benefits for the integrators because the individual services are isolated. They have a high cohesion and a low number of dependencies. In addition, in contrast to a monolithic system, the services can be integrated and deployed separately.

Testers

Before going into production, the system needs to be tested in order to identify missing functionalities and to avoid bugs. This applies to the individual services as well as the integrated system.

The microservice architectural style allows the services to be tested separately. Interface descriptions define how the services can be called, what input data they accept and what output data they produce. After the components have been put together, additional integration tests can be performed to check the system's behaviour.

While the system is evolving, updates to individual system components can happen very often. Testing efforts increase with the number of components and the complexity of the microservice architecture. Since manual testing can be very time-consuming, testers try to automate recurring tasks. This requires the service interface descriptions to be machine-readable. It also requires a repository from which executable artefacts can be obtained and deployed in an automated manner.

IT operations

Among other things, people from the IT operations group are responsible for deploying the integrated system into production, as well as configuring, maintaining, and monitoring the infrastructure.

The system deployment should be easy and fast, so that new features and updated components can be delivered in a short time. To this end, IT operations people try to leverage automation and write deployment scripts for tools such as Ansible, Chef, and Puppet. Container-based virtualisation (e.g. with Docker) allows operations people to quickly start and stop services without having to take care of software dependencies and platform requirements.

In our architecture, processing services can be put into Docker images. The JobManager is able to automatically run such services on compute nodes in the Cloud without requiring the IT operations group to explicitly install them.

IT operations are also responsible for keeping the system online without interruptions. The microservice architectural style allows them to deploy updates of individual services without restarting the whole system. It also enables Continuous Delivery strategies such as *Blue-Green Deployments* or *Canary Releases* which allow for *Zero-Downtime Releases* with the possibility to roll back faulty deployments to stable versions (Humble & Farley, 2010). Software components should be stored in a binary repository with a version control mechanism, so that older versions can be accessed and deployed quickly and in an automated manner.

In order to guarantee the smooth operation of our system, IT operations also need comprehensive logging and monitoring facilities at a centralised location. They need to be able to check the status of individual components as well as the system as a whole, and to identify operational issues such as the occurrence of an unusual number of errors, uncommon memory usage, or unexpected load peaks.

The IT operations people are also administrators who need to adapt the system *a)* to a specific infrastructure and *b)* to a certain application, domain, or use-case. The system should therefore be configurable. It should be possible to change its behaviour without modifying and recompiling the code. In our architecture we use a rule-based system whose production rules can be modified dynamically during runtime if necessary.

Business professionals

There are a number of people who have in interest in our system because they want to do business with it. For example, companies can act as resellers and offer our system to interested customers from the GIS community. These companies may provide additional products such as data and processing algorithms, or services such as the definition of pre-defined workflows. They also might want to add new software components to extend the system and to adapt it to the requirements of their customers.

In addition, there are business professionals working at data provider companies who want to offer high-quality data products to their customers and to keep costs low. These people want to make use of the Cloud in order to avoid having to maintain on-premise hardware. They also prefer a high degree of automation, so that data can be processed with minimal human interaction and hence minimal costs.

Managers of GIS projects have requirements similar to those of data provider companies. They need to carry out projects without exceeding their budgets. To summarise, business professionals require a system that has the following capabilities:

- The system should produce high quality results in a short amount of time.
- The system should be highly automatable and require minimal human interaction.
- Maintenance costs should be low. It should be easy to integrate new software components and processing algorithms in the system and to further develop them separately without having to take insight in the rest of the system.
- Costs for operation should be low. This means it should be possible to use Cloud resources instead of on-premise hardware, which would require additional maintenance efforts.
- Business professionals require a short time to market. Software extensions as well as produced data should be made available to customers as fast as possible in order to keep up with or outperform competitors.

Our system supports these requirements due to its microservice architecture, which facilitates modularity, extensibility, low maintenance costs, and a short time to market. In addition, our system has a workflow management component that allows recurring processes to be completely automated.

2.3.2 Quality attributes

In the previous section we defined stakeholders and their requirements towards our system. Based on this, we can now derive a list of quality attributes that our system should have. According to Bass, Clements, & Kazman (2012, p. 63) a *quality attribute* is a "measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders."

A common way of describing a quality attribute is by specifying a general scenario under which it becomes evident. A general scenario describes some kind of a stimulus (an event) and how the system responds to it. The scenario also includes evaluation criteria (measurable to testable) that can be used to validate if the system actually satisfies the needs of the stakeholders.

A scenario description consists of the following parts:

1. **Source of stimulus.** The actor or the system component that triggers the stimulus.
2. **Stimulus.** An event that requires a response from the system.
3. **Environment.** The circumstances under which the stimulus occurs (e.g. the state the system is in when the stimulus arrives).
4. **Artefacts.** The system components affected by the stimulus (often this is the whole system).
5. **Response.** The way the system responds to the stimulus.
6. **Response measure.** Evaluation criteria used to validate if the system responds to the stimulus as expected or required by the stakeholders.

In the following we provide a list of quality attributes our system should have. We also specify a general scenario for each attribute. In Chapter 5, *Evaluation* we will utilise the general scenarios to derive concrete ones and to evaluate our system under practical conditions.

Performance

One of the main quality attributes our system should have is a good *performance*. In Section 1.8 we presented a use case dealing with urban data that needs to be processed at least as fast as it was acquired. This means there is a time limit (or a deadline) for our system to finish the data processing—i.e. to completely execute the processing workflow. In order to satisfy requirements like this, our system should make best use of available computational resources. One way to achieve this is to distribute processing tasks to compute nodes in the Cloud in a way that available CPU power is used effectively and the amount of data transferred over the network is minimised.

The architecture presented in this chapter includes a JobManager that is able to run geospatial processes (or *processing services*) in parallel on multiple Cloud nodes. The JobManager contains a configurable rule system that can be used to add constraints for the JobManager's task scheduler and to make it leverage data locality by executing individual calls to processing services on those nodes that contain the data to be processed. In other words, the processing services are transferred to the data and not the other way around. This reduces the amount of data copied over the network and hence improves performance.

Table 2.1 describes a general scenario for the performance quality attribute.

Source	Users, GIS experts, data providers
Stimulus	Execution of a workflow
Environment	Normal operation
Artefacts	Components for workflow management—i.e. JobManager (Section 2.10) and processing services (Section 2.6)—as well as the data storage system (Section 2.7) and the network.
Response	The system executes the workflow and offers the processing results for download
Response measure	*1)* The workflow was executed in a given amount of time *2)* The system utilised available computational resources as good as possible *2.1)* All compute nodes were used to their full capacity (CPU power) *2.2)* The amount of data transferred over the network is minimised

Table 2.1 Performance General Scenario

Scalability

According to Bondi (2000) the term *scalability* means that a system should be able to "accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement". In our case this means that our system should be able to handle the following aspects:

- An increasing number of users and concurrent workflow executions
- Processing of arbitrary data volumes
- Changes to the infrastructure such as an increasing number of compute nodes

The system should perform equally well under all conditions. It should not fail under heavy load (multiple workflow executions at the same time, or huge amounts of data to process) and it should make use of available Cloud resources as good as possible. For example, if new compute nodes are added the system should be able to make use of their CPU power and execute workflows in a shorter time.

Table 2.2 summarises the quality attribute in a general scenario description.

Source	Users, data, developers, infrastructure
Stimulus	The number of a certain element increases or the volume of work grows
Environment	Normal operation, overloaded operation
Artefacts	Whole system
Response	The system continues to work
Response measure	1) The system does not malfunction under heavy load (multiple workflow executions at the same time) 2) The system should be able to process workflows faster the more computational resources are available 3) The system is able to process arbitrary data volumes without failing or becoming excessively or irregularly slow

Table 2.2 Scalability General Scenario

Availability

Our architecture executes workflows by running a high number of processing services in a distributed environment. A single workflow execution can take a long time and the users expect the system to reliably provide processing results after the workflow has finished. However, in a distributed environment there are many things that can go wrong (Nygard, 2007). For example, the communication between distributed microservices can be interrupted because of a network failure, a single service can crash because of a software bug, or a whole set of services can become unavailable because a virtual machine goes offline. Under such circumstances our system should still be able to continue operating, to execute workflows and to process data. If a workflow execution is interrupted the system should be able to resume as soon as normal operation has been restored, or to repeat or reschedule work of failed compute nodes elsewhere.

The components of our system are designed to be redundant, so there is no single point of failure (SPOF). For example, the main component of our system, the JobManager, can be run in a clustered configuration. The distributed file system we use for data storage offers a high fault tolerance with respect to hardware errors through data replication.

Source	Hardware, software, physical infrastructure
Stimulus	Crashes, lost messages, incorrect responses, timeouts, etc.
Environment	Any mode of operation
Artefacts	Whole system, virtual machines, physical infrastructure
Response	The system should prevent faults from becoming a failure. It should continue to work in a degraded mode and try to recover from it as soon as possible.
Response measure	1) The system should be still able to finish workflow execution, even if there is a fault 2) The workflow execution might take longer as usual but should produce the same results.

Table 2.3 Availability General Scenario

Modifiability

Our stakeholders require our system to be modifiable at least at the following levels:

1. The users want to control how our system processes data
2. The members of the system development team want to add new features
3. Developers of geospatial algorithms want to integrate their processing services into our system

As we will show later, our system offers a way to specify processing workflows in a Domain-Specific Language. This allows users to control the behaviour of our system in terms of what data it selects and which processing services it applies to it in what order.

Our architecture is based on microservices. Services can be interchanged and connected in different ways. The independence of individual components allows the architecture to be modified later without requiring a specific service implementation to be changed. Similarly, new services and hence new functionality may be added without modifying the rest of the system. This applies to both core services and processing services.

Source	Users, system developers, processing service developers, integrators
Stimulus	Add, remove or modify functionality or services. Change technologies, modify configurations, etc.
Environment	Compile time, build time, runtime
Artefacts	Code, interfaces, configurations, data, etc.
Response	The modification is made and deployed
Response measure	It should be possible to make modifications without having to rebuild and redeploy the whole system

Table 2.4 Modifiability General Scenario

Development distributability

As mentioned in Section 2.3.1 developers who want to contribute processing services to our system often work for different institutions distributed across many countries. Similarly, the members of the system development team can also be located in various places.

Our architecture should therefore support distributed software development. As described in Section 2.1.2 the microservice architectural style allows multiple teams to work concurrently on different aspects of the same system. Each team is responsible for one or more microservices. Since the dependencies between the services are kept at a minimum, the development can happen almost independently. The processing services that can be integrated in our system typically do not even have any dependency to any other service in our system.

Source	Developers, integrators
Stimulus	Develop system components in distributed teams
Environment	–
Artefacts	Processing services, core system services
Response	Developed and integrated software components form a system
Response measure	Independent and distributed teams can develop software components and integrate them into the system on their own

Table 2.5 Development Distributability General Scenario

Deployability

In order to put our system into production, it has to be deployed to a Cloud environment. Since a full deployment can consist of a large number of microservices that need to be distributed to multiple nodes, the deployment can be very complex and take a long time.

In order to reduce manual efforts and to make the whole process reproducible, the deployment should be automated as much as possible. In Section 5.3.6 we will evaluate the deployability of our system and will show that IT automation tools such as Ansible (Red Hat, 2017) can be used to deploy and update the complete system with only one command. In addition, our JobManager is able to deploy containerised processing services on demand using Docker, even without special IT automation scripts (see Section 2.11.4).

Source	System developers, processing service developers, integrators, IT operations
Stimulus	Deploy the whole system, update single services, or change configuration
Environment	Initial deployment, normal operation
Artefacts	Whole system, individual services, configuration
Response	The system is fully operational
Response measure	1) The deployment process is fully automated 2) All services are up and running 3) The modified configuration is in effect

Table 2.6 Deployability General Scenario

Portability

Another important quality attribute of our architecture is portability. The architecture is designed to run on various Cloud infrastructures. It does not require a specific hardware nor does it require a specific Cloud infrastructure provider such as AWS, Google Cloud Platform or Microsoft Azure. The same applies to the operating system, although we primarily tested our implementation on Linux. The core services of our system are implemented in Java and should run on Windows or other platforms as well. The majority of the processing services is containerised using Docker, which runs on various operating systems.

The only requirement is that there has to be at least one computer where we can deploy our services and perform the data processing. Whether this computer is a virtual or a physical machine is irrelevant for our architecture. Although we specifically designed it for Cloud environments our system also runs on a workstation, a Grid or a Cluster.

Portability is not only important to improve the usability of our system (in particular for IT operations) but also helps exploiting it commercially. Our system does not bind possible customers to a specific Cloud provider.

Source	Business professionals, customers, IT operations
Stimulus	The system should be deployed to a certain environment
Environment	Initial deployment
Artefacts	Whole system
Response	The system is fully operational
Response measure	The system can be deployed to multiple platforms

Table 2.7 Portability General Scenario

2.3.3 Other quality attributes

In the previous section we listed a number of quality attributes our system should have and defined scenarios which we will use later to validate that the system meets the requirements of its stakeholders (see Chapter 5, *Evaluation*).

In this section we list other quality attributes we considered while designing the architecture. These additional attributes are, however, either of minor importance for the stakeholders or their complete specification and evaluation is beyond the scope of this work. For the sake of completeness we list them here but we do not specify validation scenarios.

Usability

As described above, our system should allow GIS experts to specify processing workflows without a deep knowledge of programming or Cloud Computing. We therefore designed an interface based on a Domain-Specific Language which we will present in Chapter 4, *Workflow Modelling*. This language is intended to be easy to use.

In this work we focus on the software architecture, the data processing and the workflow modelling. In Section 4.7 we briefly describe a web-based editor for our Domain-Specific Language,

but a description of a full user interface is not part of our work. A comprehensive study on usability is also beyond the scope of this thesis.

Interoperability

Our system does not directly communicate with other systems, so interoperability is not of major importance. However, the datasets that our system processes are typically generated by other systems. Similarly, the workflow results produced by our system are usually supposed to be read by other systems. We recommend using standardised file formats for geospatial input and output data. This kind of interoperability depends almost completely on the processing services and what file formats they support but not directly on our architecture. Other services in our system do not interact with geospatial datasets.

As described in Chapter 3, *Processing* the JobManager has a rule-based system to create executable process chains from workflow definitions. With this rule-based system it is actually possible to add preprocessing and postprocessing services that convert data from one file format to another. However, this approach still depends on the file formats supported by the processing services and what conversion services are available.

In the IQmulus research project, where we created a productive system based on our design, we identified a number of file formats all processing services had to support in order to implement the use cases defined in the project. We tried to keep this number as low as possible. However, for full interoperability with other Geographic Information Systems, the services must support a wide range of file formats or there must be appropriate conversion services.

Testability

Testing is an essential part of software development. It allows developers, users, and integrators to validate if a system component (service), a set of components, or the system as a whole works as expected and meets the requirements of its stakeholders. Testing can significantly reduce the costs of integrating and maintaining a software component. It takes some effort during development to set up the tests, but it typically pays off later on, especially if the test coverage is high and the tests are automated.

In order to test a distributed system such as ours, testing should happen at various levels: classes, modules, interfaces, services, the system as a whole, etc. There are various strategies ranging from unit tests, end-to-end tests, and integration tests to acceptance tests.

Microservices are separate programs that should serve a specific purpose. Strategies such as unit testing and contract-driven testing can be used to validate if a single service meets its requirements and behaves as expected. However, as soon as many services need to be integrated to a larger application, testing becomes more complex.

A comprehensive description on how to test a large distributed system with a microservice architecture such as ours is beyond the scope of this work. For further information we refer to Newman (2015, Chapter 7). We also would like to refer the reader to Cohn (2009) and Fowler (2012) who define a concept called *test pyramid* that helps developers decide how much effort they should put into what kinds of tests.

Security

Geospatial data may contain sensitive information. Special care needs to be taken to secure the data when it is stored and processed in the Cloud. We investigated this topic in parallel with this thesis (Hiemenz & Krämer, 2018; Krämer & Frese, 2019). We summarise some relevant aspects in Section 2.13, but a comprehensive security concept is beyond the scope of this work.

2.4 Architecture overview

This section introduces the overall architecture of our system. An overview of all components is depicted in Figure 2.7. The individual components are described in detail in subsequent sections.

The GIS expert uses the system through a web-based user interface. This interface consists of three components: a data upload form, a data browser and a workflow editor.

First, the GIS expert uses the data upload form to store new geospatial data in the Cloud. The upload form sends the data to the data access service (Section 2.8) which saves it in a distributed file system (Section 2.7). In addition, a new entry with metadata about the uploaded data is created

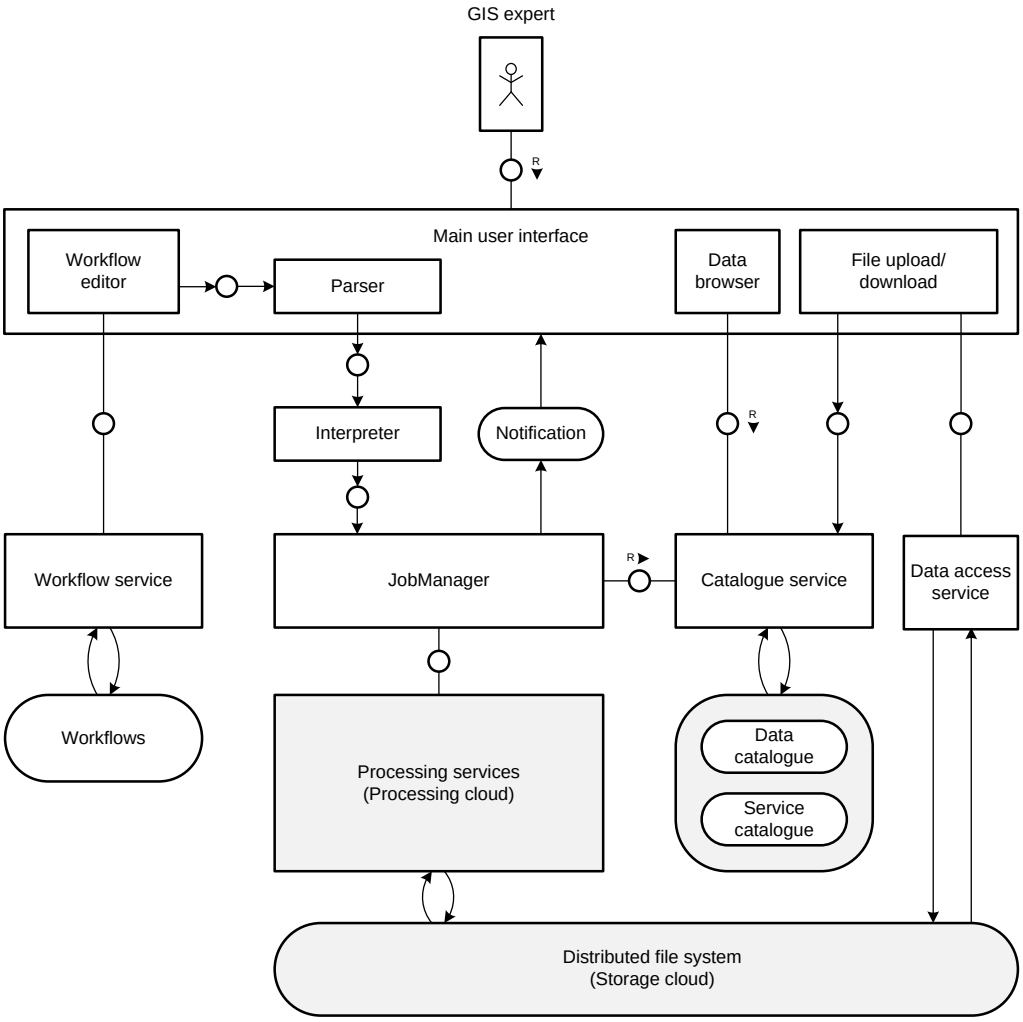


Figure 2.7 Overview of the software architecture

in the data catalogue (Section 2.9.1). Metadata can either be generated based on default values or it can be provided in the uploaded form—e.g. as an ISO 19115-1 (2014) compliant XML file.

After uploading datasets, the GIS expert specifies a high-level workflow using a Domain-Specific Language (Section 2.5). The workflow is saved in a workflow database for later use. Additionally, it can be shared with other users.

A workflow can contain placeholders for the datasets to be processed (see Chapter 4, *Workflow Modelling*). The GIS expert has to select datasets and assign them to these placeholders. For this, the GIS expert uses the data browser, which is a user interface for the data catalogue that stores all information about data uploaded to the distributed file system. The data browser allows for searching based on a spatial extent or metadata.

The GIS expert then executes the workflow through the user interface. The workflow will first be parsed (Chapter 4, *Workflow Modelling*), interpreted and finally processed by the JobManager (Section 2.10 and Chapter 3, *Processing*). The JobManager queries the catalogue service (Section 2.9) for metadata about the data to be processed (data catalogue) as well as information about the available processing services (service catalogue). The service catalogue contains information specified by the processing service developers (Sections 2.6 and 3.6.2).

After selecting the right data and processing services, the JobManager applies a pre-defined set of rules to create process chains specifying which services should be executed, in which order and on what nodes in the Cloud. The JobManager starts the services and monitors their execution while the services store their processing results in the distributed file system.

Finally, the JobManager creates a new entry for the generated result set in the data catalogue. After that, it sends a notification to the user interface. If the process is long-running and the user has already closed the user interface, this notification might also be an email sent to the user's inbox.

2.5 Workflow editor

In order to control geospatial processing in the Cloud, the GIS expert defines high-level workflows in a web-based workflow editor in our system's user interface. The editor is based on a Domain-Specific Language (DSL). The aim is that users should be able to quickly learn the DSL and to easily read and understand workflows written in it.

The main benefit of our DSL is that it is high-level and does not require users to know details about available processing services, the structure of the data stored in the Cloud, or the infrastructure the services are executed on. Instead, the users can focus on the workflow—i.e. on *what* they want the system to do and not on *how* it should be done. For example, the following workflow first selects a data set containing a recently updated point cloud. It then removes `NonStaticObjects` from the data set. `Trees` and `FacadeElements` are selected and put into another data set called `CityModel`.

```
with recent PointCloud do
    exclude NonStaticObjects
    select added Trees and added FacadeElements
    update CityModel
end
```

Note that terms such as `recent` or `NonStaticObjects` can mean many things depending on the context in which they are used (i.e. application domain). We use declarative knowledge encoded in rules to map such terms to processing services or processing parameters (see Section 4.6.3). For more information about the workflow editor, the Domain-Specific Language, and the language design process we refer to Chapter 4, *Workflow Modelling*.

2.6 Processing services

As described in Section 2.3.1 the spatial algorithms in our system are implemented by experts from various domains with different backgrounds. Each algorithm is provided as a separate program. We call these programs *processing services*. A processing service is a microservice that runs in its own process and serves a single specific purpose—i.e. it implements exactly one algorithm. For example, there is a service for corregistration, one for triangulation, one for intersecting 2D or 3D data, etc. Linking multiple processing services to a chain allows complex workflows to be created. A similar approach can be found in the Unix operating system where pipes can be used to send data through multiple programs. A workflow in our system can be compared to a Unix pipeline.

Input and output parameters of each service are described in a catalogue (see Section 2.9). This allows the system to correctly connect them to executable process chains (this process is described in Chapter 3, *Processing*). Our system supports a wide range of processing services developed using various programming languages and paradigms. This allows developers to select the best strategy to implement their services depending on the actual problem instead of the environment (i.e. our system). For example, although MapReduce is often used in Cloud Computing applications, it is not always the best solution for every problem, and other programming paradigms such as actor-based programming or in-memory computing sometimes allow for faster and more flexible algorithms. Compared to other platforms such as Apache Hadoop or Apache Spark, our system does not require algorithms to be implemented in MapReduce or a similar model. Instead, it supports arbitrary programming paradigms.

This capability of our system leads to another benefit. In the geospatial processing domain a lot of high-performance algorithms already exist and even though they might not be optimised for parallel computing it is desirable to reuse them in the Cloud instead of completely rewriting them from scratch. They can be integrated almost as-is into our system as long as they follow the guidelines described in Section 2.6.1.

To summarise, our system supports the following types of algorithms (depicted in Figure 2.8).

Single-threaded algorithms. Our architecture allows single-threaded programs (typically existing spatial algorithms) to be executed. A service with such an algorithm runs on a single compute node and uses only one CPU core. In order to parallelise the processing, the JobManager has to distribute input data to several instances of these services.

Multi-threaded/GPU algorithms. Such algorithms run on a single node only but make use of multiple CPU or GPU cores in order to increase performance. They typically scale vertically and profit from hardware upgrades—e.g. more CPUs, a better graphics card, or more memory. However, they do not scale horizontally over multiple nodes in the Cloud. In order to compensate for that, the JobManager distributes input data to multiple instances of such algorithms.

Distributed algorithms. Our architecture supports algorithms implemented using distributed programming paradigms such as agent-based programming or in-memory computing. Such algorithms are typically provided as binary executables. The JobManager executes them and oversees their resource usage.

Batch and stream processing. We use Apache Hadoop to execute batch processing algorithms implemented in MapReduce. Such jobs may be split up into multiple tasks which run on different nodes. The tasks communicate with each other through the distributed file system. The same applies to jobs implemented for Apache Spark or similar systems, as well as any stream processing framework.

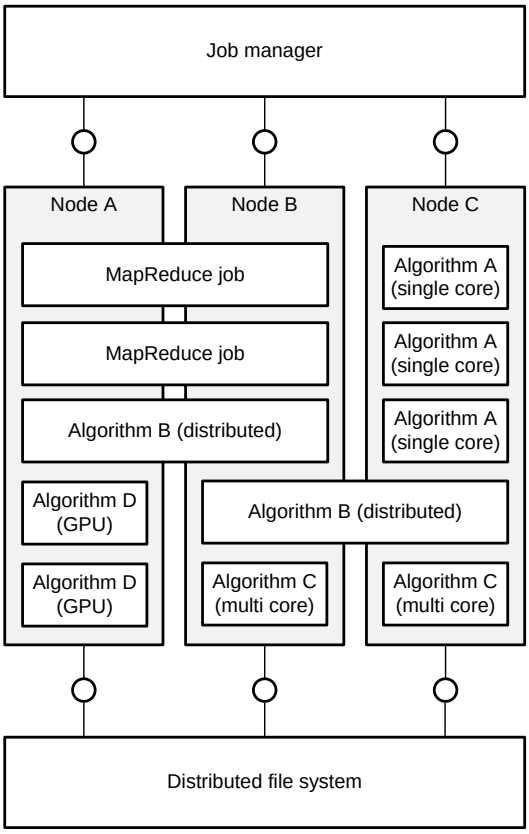


Figure 2.8 Processing services can be implemented in different ways. They communicate through a distributed file system. The JobManager executes the services in the Cloud.

Note that the most common way to implement a service is the single-threaded approach. Since the JobManager is able to parallelise service executions and to distribute data, the other approaches do not offer benefits unless a service is supposed to be integrated in various environments and not only in our system. Instead, the single-threaded approach allows service developers to focus on the algorithmics and to leave the details of distributed computing up to the system.

2.6.1 Guidelines for integrating processing services

As we will show in Chapter 3, *Processing* we do not require processing service developers to implement a specific interface in order to integrate their services into our system. In fact, one of the key contributions of our architecture is the possibility to utilise almost arbitrary processing services. In response to the requirements from the stakeholders, this particularly applies to those services and algorithms that developers have already been working on for many years or those that have been created in the context of other projects and that should now be integrated into our system.

Nevertheless, there are a few properties that a processing service should have in order to fully integrate into the concept of our architecture. These properties usually do not require fundamental modifications to existing services.

Microservice architectural style. Every processing service should be a microservice and meet the definition given in Section 2.1.2. At least it should be small, run in its own process, and serve one specific purpose.

Scientific workflow tasks. Conceptually, a processing service is a task (or a job) in a data-driven scientific workflow (see Section 3.2.1). It is a program that can be called with a number of arguments (typically a command-line application). The program reads data from one or more input files, processes the data, and writes the results to one or more output files (the locations of input and output files are part of the command-line arguments). After that, the program exits. Although we use the name *processing service* the program should not be mistaken for a continuously running web service. See Section 2.6.2 for a discussion on this type of services.

Metadata. Since we do not require processing services to implement a specific interface, the way how they have to be called can vary from service to service. In order to execute a certain service and to be able to generate a command-line call for it (with the correct number of parameters, the right labels, default values, etc.), our system's JobManager has to have information about the service's actual interface. Service developers should therefore provide a machine-readable interface description for each of their services. We call such an interface description *service metadata*. The service metadata model will be specified in detail in Section 3.6.2.

Exit code. After a processing service has finished, the JobManager evaluates its exit code in order to determine if the service execution was successful or not. According to common conventions for exit codes, a processing service should return 0 (zero) upon successful execution and any number but zero in case an error has occurred (e.g. 1, 2, 128, 255, etc.). More information on error handling is given in Section 3.7.4.

Semantic versioning. One of the requirements from the stakeholders is that processing services can be continuously developed further and that new service versions can be integrated into the system and deployed at any time. In order to ensure seamless and stable operation, processing services should be versioned according to the *Semantic Versioning Specification 2.0.0* (Preston-Werner, 2013). This specification is widely adopted and is used by many programs and software libraries. It defines strict semantics for each component of a version number. The *major* version should be changed if incompatible API changes have been made (e.g. if the service interface has changed in an incompatible way), the *minor* version should be changed if functionality has been added in a backwards-compatible manner, and the *patch* version should be changed if the service has been updated (e.g. due to a bugfix) but the interface has not changed at all.

No side effects. In order to be able to deterministically generate execution plans for workflows, the JobManager has to be in full control of the whole workflow execution. The JobManager has to know what input files a service wants to read, so it can provide them to the service in a timely manner. It also has to know which output files the service creates, so that it can pass them on to subsequent services.

Services must not have side effects. This means, for example, they must not create additional files the JobManager does not know about, nor must they read files from hard-coded locations. All input and output files must be specified as command-line arguments.

Stateless. In order to guarantee that workflow executions are deterministic and reproducible, processing service should be stateless. They should only depend on input files and other parameters given on the command-line but not on any other external state.

Idempotent. The JobManager employs strategies to handle faults during workflow execution. For example, if a processing service has failed on a compute node (e.g. because the network was temporarily unavailable), the JobManager can repeat its execution on another one. Similar to the *stateless* property, in order to guarantee deterministic and reproducible executions, processing services should be idempotent. Regardless of how many times a processing service is executed, for the same set of input files and parameters, it should always produce the same results.

Containerisation. The JobManager supports the execution of containerised processing services. A sensible technology for this is Docker (Docker Inc., 2017). Containerisation (or operating-system-level virtualisation) allows processing services to be put into separate environments. A container is like a lightweight virtual machine including a separate operating system and a file system. In our case, the major benefits of containerisation are a better separation of processes and independence of platform and system libraries. Processing services wrapped into containers are isolated. They cannot interfere with other services running on the same system. Containerised processes can only access directories and files in the virtual file system of the container. External directories and files must be explicitly mounted into the container when it is started. The mounting process is controlled by the JobManager which, as a consequence, can verify (to a certain degree) that the services do not have side effects and are stateless. In addition, since the services run in their own environment they can have arbitrary requirements in terms of operating system and library dependencies without getting into conflict with other services running on the same system.

Artefacts. Processing services and their service metadata should be put into artefacts (ZIP files) and uploaded into an artefact repository so that the JobManager can find and deploy them during workflow execution (see Section 2.11.2).

2.6.2 Other external services

As described above, processing services are expected to be command-line programs. They are launched with a set of arguments, generate a certain result, and then exit. This approach excludes continuously running web services to be used as processing services.

Nevertheless, such services can be utilised through a simple *delegate service*. A delegate service is a command-line application that performs a single request to the web service on behalf of the JobManager in the following manner:

1. It reads one or more input files,
2. sends the input data to the web service,
3. waits for its response,
4. writes the response to one or more output files,
5. and finally exits.

To the JobManager the delegate service appears like a normal processing service although it forwards (or delegates) work to another service.

The same approach can be applied to other external processing facilities. For example, Apache Hadoop or Spark jobs are typically submitted through a separate command-line application. For example, Spark applications are launched through a tool called `spark-submit` which takes a JAR file containing the actual application as well as a URL to the Spark cluster as parameters. Command-line applications such as `spark-submit` can be converted to processing services by specifying suitable service metadata.

2.7 Data storage

There are a number of possibilities how data storage can be implemented in the Cloud. In this work we focus on distributed file systems as they satisfy all of our requirements. However, for the sake of completeness, in this section we also discuss object storage and distributed databases which both offer more or less the same capabilities as distributed file systems but provide less extensive access transparency.

2.7.1 Distributed file system

A distributed file system (DFS) is a virtual file system that spans over multiple nodes in the Cloud and abstracts away their heterogeneity (see Figure 2.9). This means that the individual nodes may use different operating systems and different actual file systems, but the DFS provides a common interface for applications to access data on these nodes, without requiring them to know where the data is actually stored (i.e. on which node and in which data centre) or what operating system and actual file system is used.

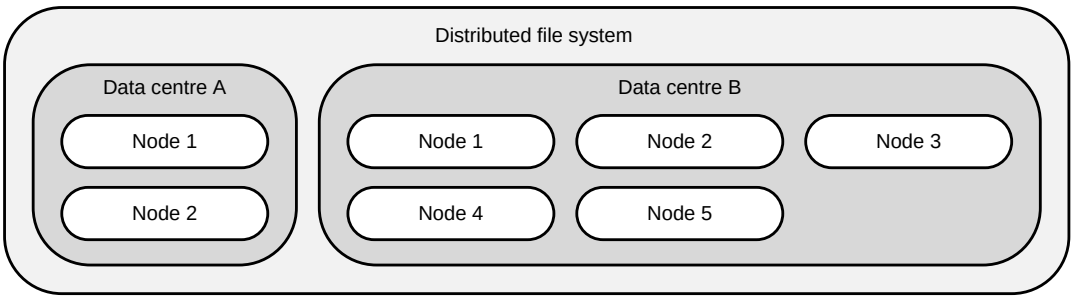


Figure 2.9 A distributed file system spans over multiple nodes possibly located in different data centres

A DFS typically has the following capabilities:

- It provides *location transparency*. Applications do not have to know where the data is exactly stored. In fact, in order to improve performance, the DFS can replicate data and create copies on multiple nodes. In Chapter 5, *Evaluation* we utilise this property to save bandwidth by executing processing services on those nodes containing the files that should be read as input data. This allows us to avoid having to transfer large amounts of data over the network. Instead, we only transfer the much smaller processing services.
- A DFS provides *access transparency*. As described above, it provides a common interface for applications to access data in a consistent way, regardless of which underlying operating system and file system is used.
- It is *fault-tolerant* since it replicates data to a configurable number of nodes. For example, if a replication factor of 3 is configured, data will not only reside on one node but be copied to two other nodes as well, so that there will be three copies. If one or even two nodes become unavailable, there will still be a third one where the data can be found.
- A DFS is *scalable* since it can operate on a small number of nodes up to a very large number. New nodes can be added on demand and, more importantly, without any downtime. Adding new

nodes typically increases available space, whereas the exact amount depends on the replication factor. Adding nodes can also increase fault-tolerance and performance of the overall system.

In our architecture, we use a distributed file system as the main data store. The DFS contains files uploaded for processing as well as workflow results. It is also used as the main communication channel between processing services. If multiple processes are called sequentially, intermediate results will be transferred through the DFS. Figure 2.10 depicts an example for this process. Service A reads an input file from the distributed file system and writes processing results back to the same file system. The subsequent service B reads the results written by A from the file system and creates another output file on the DFS.

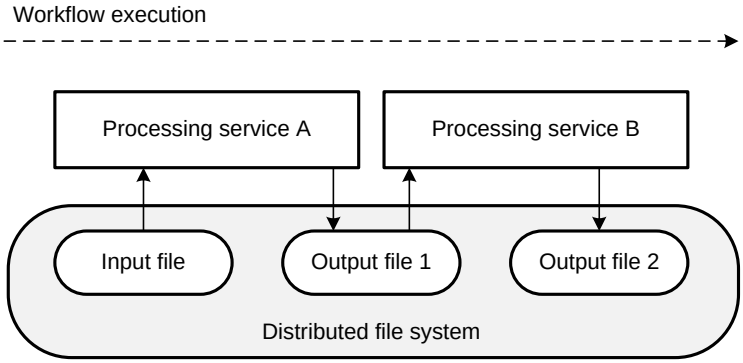


Figure 2.10 Two processing services communicate with each other through the distributed file system

If both services run on the same node or in the same data centre, the services will most likely operate on local files and do not communicate over the network. This is, however, handled transparently by the DFS driver (access transparency).

A distributed file system provides a consistent way for services to communicate with each other. The characteristics described above are essential for our architecture. They provide better maintainability, improved isolation, parallelisation, as well as the possibility to handle faults during workflow execution.

Maintainability. We do not require the processing services to implement a specific interface, nor do we require them to implement a network protocol to communicate with other services (including all aspects related to this such as handling connection problems, timeouts, or incorrect responses). They just have to be able to read and write files. This principle is also known from other distributed processing architectures such as MapReduce, for example, where map tasks and reduce tasks solely communicate with each other over files. Scientific workflow management systems such as Pegasus (Deelman et al., 2015) or Taverna (Wolstencroft et al., 2013) work similarly.

Isolation. Processing services are microservices running in their own processes. If the only way to communicate with other services is to read and write files, the processes are perfectly isolated from each other.

Parallelisation. Multiple processes can read a file at the same time without causing conflicts. This allows our system to parallelise data processing without having to implement distributed locking mechanisms. However, if multiple processes write to the same file at the same time, conflicts are very likely. Note that despite this, we do not implement write locking either. Instead, our JobManager takes care that all output file names it generates for processing service calls are unique, so that the services never write to the same file. This is also one of the reasons why the processing services must not have side effects (see Section 2.6.1).

Fault tolerance. The improved isolation allows for implementing fault tolerance. If one of the services should fail on one compute node, its execution can easily be retried on another node. If services communicated directly over the network with each other, retrying a service would require the preceding service(s) to send all input data again. In the worst case, the whole workflow would have to be restarted.

The overhead of writing results to the DFS and then reading them again in a subsequent service (as opposed to transferring them directly) is compensated by these benefits. In fact, the impact on performance is rather low. Files are in our use cases almost always processed completely and sequentially (i.e. no random file access). In addition, the underlying operating system and file system provide performance optimizations such as caches and direct memory transfer if needed.

2.7.2 Object storage

Another common way to store data in the Cloud is *object storage* where data is managed as objects as opposed to files. Object stores usually have the same properties as distributed file systems in terms of location transparency, access transparency, fault tolerance, and scalability. The main difference is that object stores are typically not mounted but accessed through an HTTP interface.

Object storage has no hierarchical structure and no way to organise objects in folders or directories like in a file system. Some object storage technologies provide workarounds to create virtual folders. For example, in AWS S3 object keys can have prefixes such as `/my/folder/` to mimic a file system hierarchy. However, S3 still has a flat structure.

The main reason why object storage is often preferred over a distributed file system is that it is typically less expensive in a Cloud environment, in particular when very large amounts of data should be stored over a longer period. In addition, it is an isolated system. This allows for separating data storage and processing, as opposed to using the same virtual machines for both.

In this work we focus on distributed file systems. However, object storage could also be integrated into our system without much effort. As we will describe in Chapter 3, *Processing* our JobManager is able to insert preprocessing and postprocessing steps into the workflow if necessary. The JobManager could insert special services that download and upload data before and after the calls to processing services. This would keep data access transparent and allow the same processing services to be executed regardless of which storage technology is used. However, it would also introduce overhead because all data would have to be downloaded before it could be processed, and the results would have to be uploaded at the end. In addition, the local hard drives of the compute nodes would have to be large enough to keep all intermediate data.

Alternatively, the processing services could be extended to support access to object storage technologies directly. However, this would contradict one of the main benefits of our approach, namely that we do not require services to implement a specific interface.

2.7.3 Databases

While traditional relational databases are often not optimised to store a large number of big data blobs, newer NoSQL technologies provide means for that. The document-oriented database MongoDB (MongoDB Inc., 2017), for example, offers GridFS, an API for storing large files inside the database similarly to a distributed file system. GridFS is very fast and offers properties such as replication and sharding and is often used to overcome limitations of traditional file systems such as maximum number of files or directories.

Access to databases typically happens through a special driver (e.g. JDBC). Since we do not require processing services to implement special interfaces, databases do not fit into our concept. However, similarly to object storage, the JobManager could insert services into the workflow that download and upload data from and to the database. This would introduce additional overhead but provide no real benefits compared to a distributed file system or object storage.

2.7.4 GeoRocket

A new technology that combines the advantages of object storage with those of databases is GeoRocket (Fraunhofer IGD, 2017). GeoRocket is a high-performance data store specifically optimised for geospatial files. It stores data in a distributed storage back-end such as AWS S3, MongoDB, HDFS, or Ceph. The data is indexed using the open-source framework Elasticsearch (Elasticsearch BV, 2017).

GeoRocket has an asynchronous, reactive and scalable software architecture, which is depicted in Figure 2.11. The import process starts in the upper left corner. Every imported file is first split into individual chunks. Depending on the input format chunks have different meanings. 3D city models stored in the CityGML format, for example, are split into `cityObjectMember` objects which are typically individual buildings or other urban objects. The data store keeps unprocessed chunks. This enables users to later retrieve the original file they put into GeoRocket without losing any information.

Attached to each chunk, there is metadata containing additional information describing the chunk. This includes tags specified by the client during the import, automatically generated attributes and geospatial-specific ones such as bounding boxes or the spatial reference system (SRS). In addition, users can set a layer path for the import allowing them to structure their data simi-

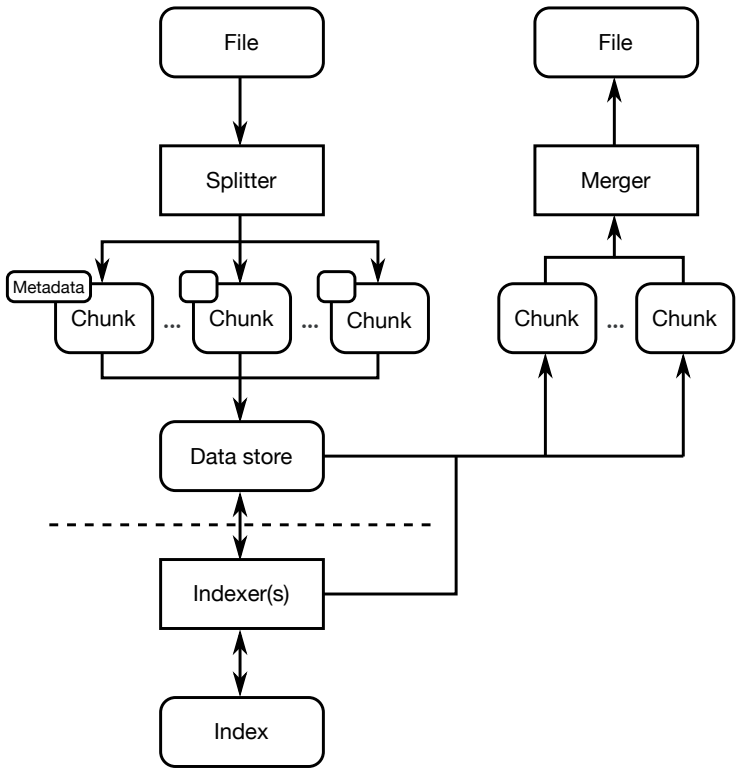


Figure 2.11 The software architecture of GeoRocket

larly to directories in a file system. Immediately after the chunks are put into the GeoRocket data store, the indexer starts working asynchronously in the background. It reads new chunks from the data store and analyses them for known patterns. It recognizes spatial coordinates, attributes and other content.

The export process starts with querying the indexer for chunks matching the criteria supplied by the client. These criteria can be specified using a flexible query language that is comparable to SQL. Matching chunks are retrieved from the data store (together with their metadata) and merged into a result file.

The main advantage of GeoRocket is that it automatically splits large geospatial data. It indexes the individual chunks and makes them accessible through a powerful query language. Compared to the other ways to store data in the Cloud, particularly the distributed file system, GeoRocket handles many aspects transparently that our architecture either requires as given (i.e. that large data sets are split into smaller parts) or has to handle itself. For example, the data catalogue we use in our architecture to store metadata on geospatial files in the Cloud (see Section 2.9.1) could be completely replaced by GeoRocket. However, similar to an object store GeoRocket has an HTTP interface. If we were to use GeoRocket in our system, the processing services would have to implement this interface. Alternatively, the JobManager could insert download and upload services into the workflow but this would again introduce additional overhead.

Using GeoRocket in our architecture is beyond the scope of this thesis. Nevertheless, despite the impact on performance, we think that GeoRocket offers enough benefits that it is worthwhile investigating its use for distributed geospatial processing in future work (see also Section 6.3).

2.8 Data access service

The data access service provides an HTTP-based interface to the distributed file system and offers operations to upload, read and delete files or directories as well as to provide file listings, set permissions etc. The service interface implements the REST (Representational State Transfer) architectural style (Fielding, 2010) and has the following characteristics:

- It is stateless
- Every file and every directory in the distributed file system is a resource and is represented by a unique resource identifier (URI)
- Responses are cacheable
- Operations are implicit and not part of the URI
  - The service uses standard HTTP methods such as GET, POST, PUT, DELETE, etc.
  - It relies on content negotiation to provide different resource representations (e.g. file listings can be rendered in HTML and JSON)

In addition, the service interface is self-descriptive and uses hypermedia links as the means of state transfer. Clients can browse the file system by entering it at the root level (the service's main entry point) and then following the links. This decouples the client from the actual URI pattern and allows clients to operate on the file system on a higher level of abstraction (i.e. hypermedia semantics). It also results in a more flexible interface that can be modified in the future without breaking clients. This capability is often referred to as the HATEOAS constraint (Hypermedia As The Engine Of Application State) and is an essential part of REST (Burke, 2013).

In order to implement high availability, several redundant instances of the data access service may run in the Cloud. Since the service itself is stateless, the instances do not have to communicate with each other. They all access the same distributed file system and therefore serve the same content.

2.9 Catalogues

In our architecture we use two different kinds of catalogues: a *data catalogue* to store metadata about the datasets in the distributed file system (the original ones as well as processing results), and a *service catalogue* keeping information about the processing services.

The data catalogue contains information such as resolution, accuracy, and the completeness of a dataset. Metadata is needed to interpret and process data in a reasonable way. The JobManager makes use of data metadata for decision making as described in Chapter 3, *Processing*. Additionally, it needs information about the processing services such as input parameters and data types to correctly execute them.

The JobManager cannot work properly if the metadata is not available. We propose to use a distributed NoSQL database such as MongoDB, which provides replication as well as automatic failover and recovery strategies. With these features the catalogues can be set up highly available and do not become a single point of failure (SPOF).

2.9.1 Data catalogue

The data catalogue service provides an HTTP interface to metadata on geospatial datasets stored in the distributed file system. The catalogue is designed to support metadata standards such as ISO 19115-1 (2014) and ISO 19119 (2016). The JobManager uses information such as resolution or size to distinguish datasets which are fast to process from others which are very detailed. It also uses this kind of metadata to decide how to split and distribute datasets to different instances of one service. If necessary, the processing services may access the data catalogue too. A more detailed description of the data metadata is given in Section 3.6.3.

2.9.2 Service catalogue

As described in Section 2.6.1, developers have to provide service metadata for every processing service they want to integrate into our system. This metadata contains information about the service and how it can be executed in the infrastructure. It is stored in a JSON file next to the service binary in the service artefact (see Section 2.11.2).

The service catalogue caches the contents of these files and provides access to them in a uniform way. The JobManager accesses the catalogue to get information about available services. It uses the service metadata to build executable process chains, to prepare the infrastructure, and to deploy the services to the compute nodes. A complete specification of the service metadata can be found in Section 3.6.2.

2.10 JobManager

The JobManager is one of the core components in our architecture. Its main responsibility is the execution of workflows in the Cloud.

Figure 2.12 shows the data flow from the workflow editor to the processing services. First, the user writes a workflow script in the Domain-Specific Language with the workflow editor. This script is then interpreted and converted to a machine-readable workflow (see Chapter 4, *Workflow Modelling*). The JobManager then converts the workflow to one or more executable process chains (see Chapter 3, *Processing*). Each process chain consists of one or more command-line calls to processing services. The JobManager assigns the process chains to the individual compute nodes in the Cloud and monitors their execution.

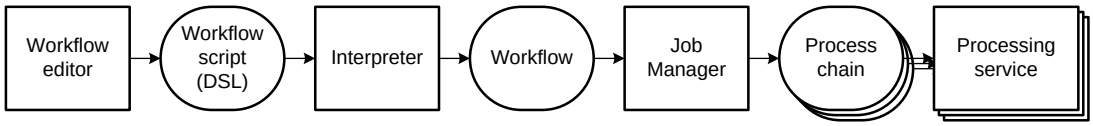


Figure 2.12 The complete process of executing a workflow in the Cloud

In order to convert workflows to process chains, the JobManager makes use of a rule-based system. This system contains production rules that specify different strategies how to distribute and parallelise the computation in the Cloud. The reasoning is based on different information including data metadata and service metadata. The whole workflow execution process in described in Chapter 3, *Processing*.

2.11 Deployment

In this section we discuss how our system can be deployed to a productive environment. We differentiate between the deployment of *processing services* and services that are core components of our system such as the JobManager or the data access service (*system services*).

2.11.1 Continuous Delivery

The term *Continuous Delivery (CD)* has been coined by Humble & Farley (2010). They describe a number of patterns that should be applied in modern software development to be able to continuously deploy software to production. This means that whenever a change has been made to the software or to one of its components, the change is automatically and immediately delivered to the customers. The main aims of this approach are to reduce the time to bring a feature or a bug fix to market, to strengthen the communication or interaction with the customers, and to continuously collect feedback to improve the software.

In order for CD to work properly, the deployment process has to be almost fully automated. The process should be reproducible and the amount of human interaction should be reduced in order to avoid errors. A common strategy is to create a *deployment pipeline* that describes the way a software artefact takes from the code repository to the production environment. A deployment pipeline is typically divided into a number of stages. The most common stages that can be found in almost any deployment pipeline are as follows (see also Humble & Farley, 2010):

- In the *commit stage* the code is compiled and validated with automated unit tests and code analysis. The steps performed in this stage are also commonly known under the term *Continuous Integration*.

- The *acceptance test stage* contains tests that validate the behaviour of the system components and whether they satisfy specified requirements.

- Some deployment pipelines contain a *manual test stage* in which human testers verify the software and try to identify defects not caught by automated tests.

- In the *release stage* the software is finally deployed into production.

It is important to note that the deployment pipeline may be aborted immediately whenever one of the steps fails. This ensures developers get instant feedback about defects in their code. It also prevents broken software artefacts from being deployed into production.

Figure 2.13 shows the deployment pipeline for our system. In our case it consists of three stages: the commit stage, the acceptance stage, and the release stage. The pipeline starts as soon as a commit is made and uploaded into a version controlled code repository.

In the commit stage we differentiate between system services and processing services. System services are compiled, tested, and deployed to an artefact repository (see Section 2.11.2). The same applies to processing services although the number of unit tests and the code analysis coverage may vary depending on the individual service developers and whether they are familiar with these concepts.

In the acceptance stage the software components built in the previous stage are first downloaded from the artefact repository. They are now treated as *black boxes* as all tests running in this stage can only access the interfaces of the components and their metadata but not their code. Again, we differentiate between system services and processing services. Defects in the service metadata of processing services can be easily detected by validating the metadata properties. Both the processing services and the system services are then tested using smoke tests. This means they are deployed to a test environment and executed at least once with pre-defined test data. The parameters for this test environment as well as the test data are stored in a version control system. Further service

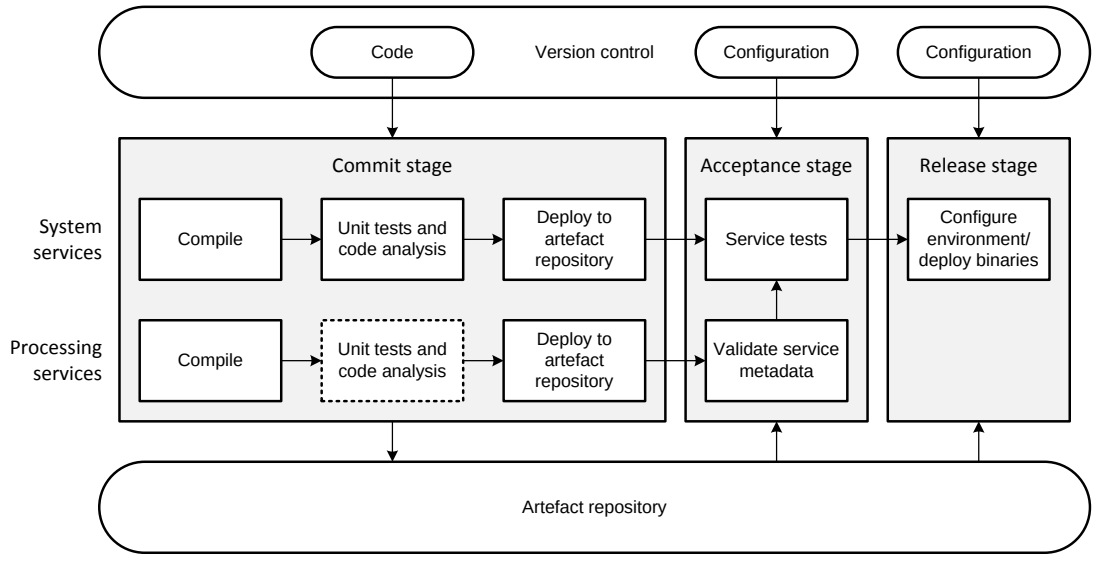


Figure 2.13 Deployment pipeline for our system with different paths for system services and processing services

tests (e.g. integration tests or end-to-end tests) may be performed in this stage. This particularly applies to system services which need to communicate with other services.

In the final stage, the release stage, the software artefacts are deployed into production. Note that the transition between the acceptance stage and the release stage is completely automatic for the processing services (see Section 2.11.4). The system services require manual interaction to trigger the release process (see Section 2.11.3). Similar to the acceptance stage, the configuration necessary to release the services to the production environment is under version control.

Version control plays an important role along the whole deployment pipeline. First, the source code of the services is stored in a code repository. Second, the built software binaries are stored in an artefact repository that also contains a version control mechanism. Finally, the configuration needed for the acceptance tests and the release process are also stored in a version control system. This approach ensures that every deployment is completely reproducible and changes to the system that introduce defects can be rolled back easily. For example, if a new version of a service is deployed to the production environment and proves to be defective (although it has passed all tests in the deployment pipeline) the previous version can be restored by reverting the change in the version control system and executing the deployment pipeline again.

2.11.2 Artefact repository

An artefact repository is a collection of software binaries and metadata. The metadata contains information about a software artefact such as its name, a human-readable description, and a version number. The repository groups artefacts with the same name but different version numbers. Once uploaded to the repository, binaries and metadata typically cannot be edited or removed—i.e. they are immutable. Due to this, an artefact repository can be compared to a version control system for binaries. Well-known products for artefact storage are JFrog Artifactory and Sonatype Nexus.

In our architecture we use an artefact repository to store system services as well as processing services and their metadata. Since most artefact repositories have a defined metadata scheme, we propose putting processing service executables and their service metadata into ZIP files and uploading them as the binary artefacts.

The artefacts are picked up by the tests in the acceptance stage of our deployment pipeline. The repository is also accessed by the JobManager in order to deploy the processing services to the compute nodes.

2.11.3 Infrastructure deployment

As discussed above, the whole deployment process should be automated as much as possible. This also includes the configuration of the environment—e.g. specific settings in the operating system on the virtual machines in the Cloud, or required system dependencies and daemons. In the DevOps movement (Loukides, 2012) the term *Infrastructure as Code (IaC)* describes the process of managing the configuration of the environment in machine-readable definition files. These files are typically stored in a version control system. They can be evaluated by IT automation tools such as Ansible, Chef or Puppet which are able to apply required configuration settings, install software and services, start daemons, etc.

The fact that the definition files are kept under version control and that they can be evaluated completely automatically allows administrators to keep a history of states of the environment and to restore a certain state with just a few commands, regardless of how many virtual machines need to be configured. In order for IaC to work properly, changes to the environment should always

be done through the IT automation tool and checked into the version control system. Manually editing the configuration of a virtual machine undermines the purpose of IaC.

In our architecture we use Infrastructure as Code to automate the release stage of our deployment pipeline. We keep configurations for the virtual machines running our system services and the processing services under version control. As mentioned above, the deployment pipeline for the system services pauses after the acceptance stage has passed successfully. The release stage has then to be triggered manually by starting an IT automation tool. This allows us to decide exactly when updates should be done to the overall system. Since the whole process can be triggered with only one command, it can be repeated several times a day. See Section 5.3.6 for more details.

2.11.4 Automatic service updates

In contrast to system services, processing services are deployed fully automatically to the production environment. The JobManager is able to download the service binaries from the artefact repository and distribute them to the compute nodes. In case the processing services are containerised with Docker, the artefact repository can store the images and act as a Docker registry. This is possible with the enterprise version of Artifactory, for example. If the free community edition is used the service artefact should contain a Docker build file (Dockerfile). The JobManager is able to automatically download an artefact and run its Dockerfile to create an image for the service on the compute node.

Note that the JobManager will only deploy services that are ready to be released. For this, it relies on the service catalogue (see Section 2.9.2). As soon as a service has passed the acceptance stage, a new entry for the specific service version will be made in the catalogue and its metadata will be transferred.

The fact that our processing services are isolated microservices allows us to deploy them independently and to make updates to our system without any downtime. This is one of the major benefits of the microservice architectural style, compared to a monolithic system that would be unavailable for the time the whole system is re-deployed.

2.12 Operations

Our system consists of a number of microservices. As mentioned earlier, in Section 5.3.6 we show that in the IQmulus project we deployed more than a hundred services to the production environment. Most of these services are started multiple times. For example, the processing connector service (see Section 3.7.5) has to be installed on every compute node. Likewise, during a workflow run multiple processing services are spawned at the same time on the individual compute nodes. The number of running service instances at a given point in time can therefore be much larger than the total number of services.

The fact that all these services are distributed across several virtual machines can make it very hard to maintain an overview and to ensure smooth operation. In this section we discuss two operational aspects that help get an insight into the running system: monitoring and logging.

2.12.1 Monitoring

In order to be able to manage a large number of distributed services, IT operations not only need to have an overview of which service instances are currently running but also about their state,

CPU usage, memory usage, etc. Maintaining an overview and monitoring the services is key, in particular when there is an issue with the system that needs to be found or if developers and administrators try to identify performance bottlenecks. Similarly, IT operations also needs to have in-depth information about the resource usage in the Cloud. For this, they have to monitor the virtual machines and collect metrics about available memory, free hard drive space, etc.

IT operations often make use of dashboards to display current metrics about the infrastructure. Viewing numbers from a single point in time is, however, typically not enough to get a clear picture about a system's behaviour. Monitoring tools therefore collect time series of data to display historical information. As Nygard (2007) has noted, it is indeed possible to predict how a system will behave in the future by looking at how it did in the past. Collecting metrics over a period of time is therefore not only useful to get a system's current or earlier state, but also helps make assumptions about how it will react to future events.

Monitoring tools for distributed systems have been in use for quite some time. One of the most mature and well-known products is Nagios (Nagios Enterprises, 2017). This tool allows for collecting common system metrics such as CPU, memory, or hard drive usage. It can also be used to monitor the state of individual services—e.g. by performing health checks. Nagios only allows for *black-box monitoring*. This means it cannot collect metrics about the internal state of a distributed application. For example, it does not know about how many connections are currently in use from a service's internal database connection pool, or how many garbage collection cycles have been triggered over the last minute.

In contrast to the black-box monitoring of Nagios there is *white-box monitoring* which describes the approach of collecting additional service-internal or application-specific metrics. One of the solutions implementing this approach is Prometheus (Prometheus Community, 2016). The open-source tool runs as a separate service which frequently collects metrics from other services in the system. Besides CPU, memory and hard drive usage, common metrics recorded with this tool are the number of open HTTP connections, request latency, open file handles, but also information about the number of failures occurred. Metrics like these can be of great importance for developers and system administrators to understand what is actually happening or has happened in the system.

Internally, Prometheus is a time-series database. It keeps a history of metrics which can later be queried with a powerful expression language. There are tools that work on top of Prometheus and offer additional functionality. Grafana, for example, is an open-source product that can be used to create live dashboards showing collected metrics in various ways (Grafana Labs, 2017). For our system evaluation which we will present in Chapter 5, *Evaluation* we used Prometheus

Figure 2.14 Screenshot of a Grafana dashboard showing metrics collected during a workflow run

in combination with Grafana to collect metrics about the behaviour of the JobManager, the processing services, the compute nodes, etc. in various scenarios. Figure 2.14 shows a screenshot of a dashboard we created with these tools. More detailed figures will be presented later.

2.12.2 Logging

Another way of performing white-box monitoring is logging. In contrast to mere numerical metrics, log files may contain additional textual information and give a broader context on a certain aspect of the system. Logging is one of the oldest and well-known methods to monitor an application or a distributed system. It is also the method providing the best loose coupling. Compared to metrics collected with solutions such as Prometheus, logging does not depend on a specific product. Log files are plain text files and can be processed by any tool or framework.

In a microservice architecture with more than a hundred services distributed to several Cloud nodes, it is important to maintain an overview of all log messages. In recent time, the so-called *ELK stack* (or *Elastic Stack*) has become the de-facto standard for distributed logging. It consists of three tools: Elasticsearch, Logstash and Kibana (Elasticsearch BV, 2017). Logstash is a service that collects, parses and transforms log messages. Most logging frameworks can be configured to push messages into Logstash. The tool sends transformed log messages to downstream services for further processing. One of these services is Elasticsearch which maintains an index of all collected messages. Elasticsearch offers a query language with which individual log entries can be found and aggregated. The query results can be displayed in Kibana. This tool offers a web-based graphical user interface that can be used to browse through log files, perform queries and create graphs.

2.13 Security

In order to protect the rights of owners of the geospatial data stored and processed by our system, various security issues have to be considered. As mentioned above, a comprehensive security concept is beyond the scope of this work. Nevertheless, in this section we give a brief overview of the most important aspects and how they can be addressed with our architecture.

Data storage. A typical way to protect data against unauthorised access is data encryption. As described in Section 2.7 we use a distributed file system as the means to store geospatial data redundantly and in a distributed manner in the Cloud. Most distributed file systems do not offer data encryption out of the box. Such a feature would have to be implemented on top of the file system as a separate layer.

Alternatively, a more sophisticated data storage solution could be used. In parallel with this thesis, we investigated approaches to secure data storage in the Cloud. We used GeoRocket as a data store (see Section 2.7.4) and implemented Searchable Symmetric Encryption (SSE) on top of it (Hiemenz & Krämer, 2018). In addition, we were able to show that geospatial data can be kept securely in an object store in a hybrid public/private Cloud environment, while still allowing the data to be processed and shared with third parties (Krämer & Frese, 2019).

Note that in a private and trusted Cloud, data encryption typically has no benefits but imposes performance and development overhead.

Data transfer. In a distributed system, data is typically transferred between multiple nodes. Often it even has to leave a data centre and has to be copied to another one. If security is important it should be made sure that communication between distributed microservices is encrypted. Since most of our services use HTTP, this can be easily realised by enabling SSL/TLS and switching

to HTTPS. In addition to the services, the distributed file system should also be configured to encrypt data transfers between Cloud nodes. Solutions such as GlusterFS or HDFS support network encryption out of the box and just need the correct configuration items to be set.

Data upload and download. The data access service (see Section 2.8) is a REST service that enables access to the data stored in the distributed file system through a defined HTTP interface (either from a client application or through the Web Browser). Access to this service can be secured through HTTPS using SSL/TLS.

Authentication. In order to protect our system against unauthorised access, an additional authentication layer can be added. Since our system consists of many microservices we propose to use a Single Sign-On solution such as the open-source tools Keycloak and CAS (Central Authentication Service). Such an approach makes it easier to interact with the individual distributed services in our system as users have to authenticate only once at a central location. Single Sign-On systems generate tokens that can be passed from one service to another to transport user authentication information.

Authorisation. While authentication makes sure only the right people have access to the system, it does not protect individual data sets stored in the distributed file system against unauthorised access. Most distributed file systems implement a permission model similar to the one found in the UNIX operating system. Each file is associated with an owner and a group. Separate read/write permissions for each file and directory can be assigned to the owner, the group, or all other users. This allows users with the right permissions to protect their files against other users who are authenticated but not authorised to access the data.

2.14 Summary

In this chapter we presented our architecture for the processing of large geospatial data in the Cloud. We discussed relevant background and related work. After that, we performed a requirements analysis and formulated quality attributes. We then presented details on the individual components of our architecture. We also discussed deployment, operations and security.

One of the main goals of our architecture is to provide GIS users and developers with access to the Cloud. Our approach to achieve this goal is based on the microservice architectural style. Compared to a monolithic application, a microservice architecture is more modular, maintainable and extensible. In our case, developers and researchers from the geospatial community can contribute processing services and extend the overall functionality of our system. There are only minimal requirements that the processing services need to satisfy. Existing algorithms can be reused in our system without fundamental modifications. Since our workflow management component, the JobManager, is able to automatically deploy processing services to compute nodes and to parallelise their execution, our architecture even allows developers and researchers who do not have an IT background or knowledge of distributed programming to leverage the possibilities of Cloud Computing.

Due to the modularity and extensibility of our architecture, a broad range of processing services can be integrated in order to create a Cloud-based system that covers a functionality similar to a desktop GIS. With the workflow editor based on a Domain-Specific Language users can automate recurring tasks and harness the capabilities of the Cloud in terms of storage and computational power. In the following two chapters we follow up on this and discuss workflow-based data processing as well as workflow definition in detail.

3 Processing

The focus of the previous chapter was on the overall software architecture of our system for the processing of large geospatial data. We described requirements and system parameters and introduced the individual components.

In this chapter we focus on distributed data processing and workflow management. The main contributions are in the way we integrate and orchestrate services based on lightweight service metadata. In terms of workflow management we contribute to the state of the art by presenting an approach to dynamic workflow execution that does not rely on a priori design-time knowledge (compared to existing workflow management systems that require all parameters to be known in advance). This approach is based on configurable rules that select processing services and datasets, and generate executable process chains leveraging data locality.

The chapter is structured as follows. We first introduce the JobManager service which is in our architecture responsible for workflow management. We describe requirements it has to meet and compare it to related work. After that, we focus on the JobManager's software architecture. We describe the individual components within the service, as well as the control-flow between them and their interfaces. We also cover aspects such as fault-tolerance, elasticity and scalability. Finally, we evaluate the JobManager's functionality against a number of patterns often found in Workflow Management Systems. The chapter concludes with a summary.

3.1 Introduction

A Geographic Information System (GIS) typically offers a range of spatial operations such as creating intersections and unions, buffering, or more advanced features like co-registration or data fusion. Users working with a desktop GIS, for example, usually import some data into the system, apply one or more spatial operations and save the modified result back to their hard drive or a database. Since large geospatial datasets often consist of many files that should be modified or processed in a similar way, Geographic Information Systems offer ways to automate work. For example, the open-source software QGIS has a Python API that can be used to create scripts that can be applied to a number of files in a batch. This can help GIS users to save a lot of time.

The approach of automating recurring tasks is well-known in computer science and has a long history (Ludäscher, Weske, McPhillips, & Bowers, 2009). A Workflow Management System is a

software that allows for modelling business or scientific processes (or workflows). Such a system typically also offers a way to automatically execute modelled workflows. A workflow is a chain of activities. In the case of scientific workflows (see Section 3.2.1) an activity is a processing step that is applied to input data and produces output data.

A GIS with scripting and batch-processing facilities can therefore be compared to a simple Workflow Management System. However, the more often users have to process data and the larger the datasets become, the more important it is to fully automate processing workflows and to avoid human interaction. In addition, desktop-based Geographic Information Systems run on a single computer and do not leverage the possibilities of distributed computing.

In this chapter we present a component called JobManager which can automatically execute geospatial processing workflows in a distributed environment. Together with the Domain-Specific Language we will present in Chapter 4, *Workflow Modelling*, the JobManager forms a Workflow Management System. In order to execute the workflows, we leverage the Cloud.

Many Workflow Management Systems are suitable for a certain kind of use case or application domain. This limits their use, as they cannot be applied to problems outside the targeted application domains. They also often have very specific requirements towards the environment they run in, the infrastructure they can be deployed to, as well as the data they can handle (see Section 3.3.1). Our JobManager, on the other hand, is designed to be more flexible. We employ a production rule system (or rule-based system, or expert system) which allows us to configure the workflow execution and to adapt it to various needs and conditions. This allows us to apply our workflow management component to many use cases and to deploy it to various infrastructures (see Chapter 5, *Evaluation*).

Since geospatial datasets can become very large and complex, the processing workflows can take a long time, ranging from a couple of hours to several days or weeks. If the process is interrupted—e.g. due to a system failure—it is important that it can be resumed without information loss and without the need to repeat work that has already been done. This is particularly important in a distributed environment where failures are expected to happen (Robbins, Krishnan, Allspaw, & Limoncelli, 2012). We therefore designed the JobManager to be resilient and fault-tolerant (see Section 3.8).

The fact that geospatial data sets can be large and complex also requires a scalable system. We designed the JobManager to be able to handle arbitrarily large datasets. It utilises available Cloud resources but can also scale out dynamically if necessary (see Section 3.10).

3.2 Background

Before we compare the JobManager to related work and describe its components in detail, we clarify what kind of Workflow Management System it actually represents. We also introduce patterns that are often found in Workflow Management Systems. We will later make use of these patterns to compare our approach to related work and to qualitatively evaluate it in terms of functionality and capabilities.

3.2.1 Business workflows and scientific workflows

Workflow Management Systems have a long history in computer science. They have been used for many use cases ranging from simple office automation to complex tasks such as the modelling of business transactions or even genome sequencing. A distinction is made between Workflow Management Systems for business modelling (*Business Workflow Systems*) and those that are used

to process information in a scientific context (*Scientific Workflow Systems*). There is an overlap between these two kinds of systems, but one can summarize the major differences as follows (see also Ludäscher et al., 2009):

- Business workflows model processes in a company and often involve humans, whereas scientific workflows are typically completely automated and executed without any human interaction.
- While business workflows model the flow of control between actors (e.g. humans, departments or—if the workflows are automated—web services), scientific workflows represent the flow of data (e.g. how a specific data entity is passed from one workflow task to another through output and input interfaces).
- The outcome of business workflows is typically known at the time of modelling. The main goal is to provide a better understanding of the process for all involved parties. Scientific workflows, on the other hand, are more experimental. They start with a given set of information (or a hypothesis) and produce a result. What the result actually is can only be determined by running the workflow.
- In business workflows there can be many independent tasks and activities happening at the same time. This helps model the complex ecosystem of an organisation. Scientific workflows, on the other hand, are typically a set of tasks that are connected to a directed graph or a process chain where data flows or streams from the beginning (the first task) to the end (the last task in the chain).

In this work we focus on data-driven scientific workflows as they match our use cases and requirements better than business workflows.

3.2.2 Workflow patterns

In order to be able to assess existing Workflow Management Systems, one has to compare their features and the different kinds of workflows they support. For lack of a "universal organisational theory" that could be used for this purpose, van der Aalst et al. investigated various Workflow Management Systems over the course of 15 years and collected a number of what they call *workflow patterns* (Russell, ter Hofstede, Edmond, & van der Aalst, 2004b, 2004a; Russell, ter Hofstede, van der Aalst, & Mulyar, 2006; Russell, van der Aalst, & ter Hofstede, 2016; van der Aalst, ter Hofstede, Kiepuszewski, & Barros, 2003). We will use these patterns in Section 3.3.1 to compare our work to other Workflow Management Systems. In addition, in Section 3.11 we describe which patterns our system supports and how we implement them. There are three kinds of workflow patterns:

- *Control-flow patterns* describe the features a Workflow Management System offers to define the sequence in which individual actions in a workflow are executed.
- Workflow *resource patterns* relate to the way a system makes use of available resources to execute tasks. In our case, this means these patterns describe how the individual tasks in a workflow are assigned to compute nodes in the Cloud and how their execution is triggered.
- *Data patterns* deal with data access and transfer.

Whenever we refer to those patterns we use the same naming as in the original work. *Control-flow patterns* start with the prefix *WCP-* (Russell et al., 2006; van der Aalst et al., 2003).

Workflow *resource patterns* start with *R-* (Russell et al., 2004b). *Data patterns* are only numbered in the original work (Russell et al., 2004a) and do not have a prefix. In order to differentiate them from the other ones, we use the prefix *D-*.

For detailed descriptions and a complete list of workflow patterns, we refer to the work of Russell et al. (2016). However, a set of patterns that is notable and relevant to our work is WCP-12 to WCP-15. These patterns relate to the capability of a Workflow Management System to run multiple instances of an activity depending on certain a priori knowledge. *WCP-12 (Multiple Instances without Synchronization)* describes the general capability of running multiple instances of an activity in parallel. Systems that implement pattern *WCP-13 (Multiple Instances with a priori Design-Time Knowledge)* are able to create multiple instances whereas it is already known at design time—i.e. when the workflow is defined—how many instances should be created. *WCP-14 (Multiple Instances with a priori Run-Time Knowledge)* describes systems that can create multiple instances of an activity even if the number of instances is only known during run-time of the workflow—e.g. if it depends on the result of a previous activity. Systems supporting *WCP-15 (Multiple instances without a priori run-time knowledge)* are able to adapt the number of instances while the activity is running. This means they allow new instances of an activity to be created even when some instances are already being executed or have completed.

It is important to note that while it is relatively straightforward to implement WCP-13 with a priori design-time knowledge, depending on the actual implementation it can be very hard to synchronise results of multiple instances of an activity if WCP-14 or WCP-15 should be implemented. Many Workflow Management Systems therefore do not support multiple instances without a priori design-time knowledge (van der Aalst et al., 2003). The system we present in this chapter, on the other hand, supports multiple instances with a priori Run-Time Knowledge (*WCP-14*) and can therefore be applied to a wider range of use cases.

3.3 Related work

In this section we put our approach in the context of related work. The JobManager executes and monitors user-defined workflows in the Cloud. It can be compared to a Workflow Management System. We list some of these systems supporting scientific workflows in a distributed environment in Section 3.3.1 and compare them to our approach. The JobManager's Rule System evaluates workflows and generates executable process chains. For this, it has to orchestrate processing services with different interfaces. We discuss service and component orchestration in Section 3.3.2.

3.3.1 Workflow Management Systems

There are a lot of Workflow Management Systems covering a wide range of use cases (Sheth, van der Aalst, & Arpinar, 1999). In this section we specifically focus on those supporting scientific workflows and operating in a distributed environment (i.e. a Cloud, Grid or Cluster).

One of the most popular Workflow Management Systems targeting science applications is Apache Taverna (Wolstencroft et al., 2013). The open-source system has been used in production for many years. It specifically targets use cases from the area of bioinformatics but can also be used in other related domains. Taverna has a graphical workflow editor that runs as a desktop application. Once a workflow has been modelled, it can be executed either locally or on a Grid, Cluster or Cloud infrastructure. The system supports various service types. It can execute web services that offer a WSDL interface (W3C, 2001), REST services, bean shell scripts and others. It also supports executing command-line applications (or *tools*), for which users have to specify

input and output ports. This is in contrast to our approach where the interface of a service is described by the service developer and not by the user.

The data items passed from one task in a Taverna workflow to another are either primitive values or lists (workflow data patterns *D-9* and *D-27* to *D-31*). Command-line tools read their input from the command line or from files and write results to the standard output stream or a new file. Taverna does not support exclusive choices (workflow control-flow pattern *WCP-4*) or arbitrary cycles (*WCP-10*). However, it supports structured loops (*WCP-21*) with post-conditions. This is typically used to repeatedly call a web service until it returns a certain result.

The possibility to specify multiple instances of a workflow task without a priori design-time knowledge (*WCP-13* and *WCP-14*) is limited in Taverna. The system has a notion of *implicit iterations*. If task A produces a list but a subsequent task B expects a single value as input, B will automatically be executed multiple times for each list item. This works for web services and beanshell scripts, but not for command-line tools that write to the standard output stream or to a single file. The processing services we use in this work can write results to multiple files in a directory. Our JobManager supports iterating over the files in this directory and calling subsequent services for each of them. In Taverna such a use case is only possible with a workaround. A service can be wrapped by another one that writes the absolute path of the directory containing the results to a separate file. A subsequent task can then list the contents of this directory and output a list of files. Through the implicit iteration mechanism, multiple task instances can then be applied to the items on this list.

This approach makes the workflow unnecessarily complex. For geospatial applications where this case happens rather often (e.g. see the description of the workflow of our use case B in Section 5.2.2) the workaround is impractical. In addition, the fact that the wrapper service has to write an absolute path to a file, binds the workflow to a certain execution environment—i.e. one that has a local file system. In general, Taverna does not focus very much on issues of portability. Workflows that have been designed for a certain platform can often not be transferred to other environments easily without changing the tasks or the executed services.

Another Workflow Management System that specifically aims for platform-independent scientific workflows is Pegasus (Deelman et al., 2015). Just like Taverna, Pegasus is open-source and has been under constant development for many years. Its origins are in bioinformatics, but Pegasus has been used for many other use cases from various domains. The system supports executing workflow tasks on various platforms including Grids, HPC clusters, and the Cloud. It relies on HTCondor which is a mature system for high-throughput computing on distributed platforms (Thain, Tannenbaum, & Livny, 2005). The system handles issues of portability by abstracting datasets, workflow tasks and execution sites from the actual files, processing services and runtime environments respectively. Pegasus has a replica catalogue containing information on how to look up files, a transformation catalogue specifying how a workflow task should be executed, and a site catalogue containing information about the computational and storage resources. This is very similar to the service metadata and data metadata in our JobManager (see Sections 3.6.2 and 3.6.3).

Workflows in Pegasus are Directed Acyclic Graphs (DAGs). The main exchange format for workflows is XML. As the name implies, Pegasus DAGs must not contain cycles (*WCP-10*). There are no conditions (*WCP-4*) and therefore no structured cycles (*WCP-21*). However, Pegasus compensates this drawback by offering an API for popular general-purpose programming languages including Python and Java. The API provides a means to generate DAG XML files on a high-level. Since the general-purpose languages have loops and conditional expressions, complex workflows can still be created. However, the whole workflow structure must be known at design time (*WCP-13*). The possibility to determine the number of instances of a workflow task at runtime (*WCP-14*) is not directly possible. Instead, one has to create a task that generates a new DAG XML file for a subworkflow, plus another task that submits the new subworkflow to the system's scheduler. This is one of the major differences between Pegasus and our JobManager.

Another difference is in the way both systems handle files and computational resources. The workflow tasks in both Pegasus and the JobManager communicate with each other by reading input from files and writing results to new files (*D-29*). In Pegasus these files can either be located on a distributed file system, an object store such as AWS S3, or other storage systems. Pegasus is able to automatically handle various storage types and provide a transparent way for the workflow tasks to access the data. The resource management features are directly integrated into Pegasus. The JobManager, on the other hand, is more lightweight and distributes work to other microservices. For example, if a certain processing service is not able to read a file from a distributed file system or from an object store, the JobManager's Rule System (see Section 3.7.3) can insert an additional service into the workflow that downloads the requested file to the local file system, and another one that later uploads the processing results back to the data store.

Both systems support allocating computational resources based on algorithms such as Random Allocation (workflow resource pattern *R-RMA*) or Round-robin Allocation (*R-RRA*). The JobManager additionally supports allocating based on capabilities (*R-CBA*) and shortest queue (*R-SHQ*) while Pegasus supports the HEFT algorithm which prioritises tasks and selects processors which minimise their finish time (Topcuouglu, Hariri, & Wu, 2002). In order to leverage data locality—i.e. to reduce the amount of data that needs to be transferred between two jobs and to optimize the workflow's execution time—the JobManager can execute multiple workflow tasks on the same computational resource. The JobManager's Rule System generates process chains which are a set of processing services executed on the same virtual machine. This capability is described in workflow resource patterns *R-RF* and *R-CE*. In Pegasus workflow tasks can be grouped to be executed on the same site. Pegasus also offers a way to cluster tasks, although this feature is typically only used for short-lived tasks to reduce the latency introduced by the HTCondor scheduler.

Gil et al. (2011) describe an extension to Pegasus, which is called WINGS. This extension allows users to design semantic workflows that are independent of the technical details of their execution. WINGS contains a rule-based semantic reasoner that is able to transform a semantic workflow to a concrete one by selecting matching data sources and workflow task instances. This approach is similar to ours, since we also propose a technology-independent way (i.e. our Domain-Specific Language described in Chapter 4, *Workflow Modelling*) to define an abstract workflow (see Section 3.6.1), which is translated to executable process chains by a rule-based system (see Section 3.7.3). However, the semantic reasoner in WINGS makes use of ontologies and requires all datasets and task instances to be specified using OWL (Web Ontology Language). Our approach, on the other hand, is more lightweight and allows arbitrary datasets to be processed, even if they are not described by an ontology. Our approach also does not require additional efforts from users and workflow task developers to learn OWL and to define missing ontologies.

Another Workflow Management System worth mentioning is Kepler (Ludäscher et al., 2006). This system is mature and has been used for various use cases for many years. Although there is an initiative to harness the possibilities of distributed computing in the *Distributed Execution Interest Group*, Kepler currently only works on a local computer and has not been transferred yet to the Cloud. Nevertheless, similar to Taverna, Kepler allows remote web services to be queried which enables distributed computing to a certain degree. For example, Jaeger et al. (2005) present an approach to use Kepler for the composition of web services to perform geospatial analysis and environmental modelling. They conclude that Kepler is a convenient system for discovering and composing web services. Our JobManager, on the other hand, is more flexible than Kepler and is specifically designed to run in a distributed environment.

A novel approach to distributed workflow execution is presented by Balis (2014). He introduces the HyperFlow workflow engine, a lightweight application written in JavaScript/Node.js. HyperFlow has a workflow model consisting of *functions* and *signals*. Functions are written in JavaScript and have no dependencies other than the Node.js environment. In contrast to other Workflow Management Systems, HyperFlow functions do not have input and output ports. Instead, they

exchange data through signals which are asynchronous events. HyperFlow supports for-each loops (*WCP-21*), choices (*WCP-4*), splits (*WCP-2*) and joins (*WCP-3*).

The fact that HyperFlow uses lightweight JavaScript functions makes it suitable for running in a Cloud environment and helps leverage the novel *serverless* paradigm. *Serverless computing* (also known as *Function as a Service* or *FaaS*) is a way to execute code in the Cloud without requiring the user or software developer to maintain a virtual server. Serverless computing even works without containers as the whole life-cycle of the executed function is managed by the Cloud provider.

Malawski (2016) has used HyperFlow to run the Montage workflow (Jacob et al., 2009) on the Google Cloud Functions platform. His experiments confirm that serverless infrastructures can be used to run scientific workflows. However, he concludes that there are some limitations that need to be considered. Cloud providers typically limit the runtime of a serverless function (to 300 seconds in the case of AWS Lambda, for example, and to 60 seconds for Google Cloud Functions). Task granularity plays an important role. In our case, serverless functions are not applicable, because many of the geospatial processing services we execute can take longer than the limits set by the Cloud providers. In addition, the number of programming languages supported by serverless computing platforms is limited. They also require the functions to implement an interface which is typically vendor-specific. One of the major goals of our work, however, is to support arbitrary services and to avoid specialised interfaces.

In addition to the related work presented here, there are many other Workflow Management Systems, each of them having a different set of features and targeting different domains or use cases (Sheth et al., 1999). Manolescu (2000) assumes there are so many systems because they often do not match the requirements of developers who want to leverage workflows in their software, or because they make assumptions that do not hold later. This is confirmed by Ludäscher et al. (2009) who state: "Given the vast range of scientific workflow types […] there is no single best or universal model of computation that fits all needs equally." In this section we have presented a selection of the most popular approaches to distributed management of scientific workflows. For an overview of the differences between further systems we refer to Yu & Buyya (2005) and Deelman, Gannon, Shields, & Taylor (2009).

3.3.2 Service and component orchestration

In order to connect geospatial processing services, we define a lightweight service metadata model (see Section 3.6.2) which can be used to specify inputs and outputs as well as other information about a service. The JobManager's Rule System uses this metadata to decide if two services are compatible to each other and if a valid process chain can be created for a given workflow (see Section 3.7.3).

Similar work has been done in the area of component orchestration. The Common Component Architecture (CCA), for example, defines a common interface description language for components used in scientific computing (Armstrong et al., 2006) named *Scientific Interface Definition Language (SIDL)* (Cleary, Kohn, Smith, & Smolinski, 1998). The Object Management Group has standardised the *CORBA Component Model (CCM)* (OMG, 2006). In addition, there is the *Grid Component Model (GCM)* which specifically targets applications running on large-scale heterogeneous Grid infrastructures (Baude et al., 2009). These models have been evaluated and used for Cloud computing applications by Malawski, Meizner, Bubak, & Gepner (2011). They describe how components of a distributed application can be packaged into platform-independent virtual appliances. Malawski et al. propose an API based on the general-purpose programming language Ruby with which components can be orchestrated to a scientific workflow.

There are other interface description languages from the area of Service-Oriented Architectures. The *Web Service Definition Language (WSDL)*, for example, describes how web services can ex-

change messages based on XML (W3C, 2001). The Workflow Management Systems Taverna and Kepler support WSDL services (see Section 3.3.1). The *Web Services Business Process Execution Language (WS-BPEL)* leverages WSDL and provides an XML-based language for the modelling of business processes (OASIS, 2007). WS-BPEL can be used for web service orchestration and choreography—i.e. to model executable processes and to track message sequences between parties and sources (Barros, Dumas, & Oaks, 2006; Peltz, 2003).

In the geospatial community there are efforts to standardise the way web services can be orchestrated to implement distributed geospatial processing. The *OGC Web Processing Service (WPS) Interface Standard* defines rules for the description of inputs and outputs of geospatial processing services (OGC, 2015). This standard has been used to facilitate distributed applications running in the Web or on a Grid infrastructure (Friis-Christensen, Lucchi, Lutz, & Ostländer, 2009; Lanig, Schilling, Stollberg, & Zipf, 2008; Rautenbach, Coetzee, & Iwaniak, 2013; Stollberg & Zipf, 2007). Another application is presented by de Jesus, Walker, Grant, & Groom (2012) who have used the Taverna Workflow Management System to orchestrate WPS services and to analyse digital elevation models in a distributed environment. Compared to WPS, our approach is more lightweight and flexible. It allows for executing arbitrary processing services and not only web services.

Patil, Oundhakar, Sheth, & Verma (2004) have noted that in order to automatically discover web services that meet the requirements of a certain use case or process, mere interface descriptions are often not sufficient. Instead, they propose to add semantics as they are offered by ontologies. They argue that semantics helps them categorise services into application domains in order to improve the service discovery process, especially if users have to deal with thousands of services. However, they also note that the process highly depends on the quality of the selected ontologies. The approach for service selection we present in this chapter does not use ontologies. Instead, it uses a lightweight mechanism based on metadata and configurable rules.

In our approach, we use a rule-based system to generate executable process chains from workflow definitions. In the area of web service orchestration, rule-based systems have been used for business workflows, mostly to implement business rules (Charfi & Mezini, 2004; Weigand, van den Heuvel, & Hiel, 2008). They have also been used to discover services and resources (Lutz, Lucchi, Friis-Christensen, & Ostländer, 2007). Compared to previous work, the rule-based system in our JobManager is used to a higher degree for multiple purposes. It selects services and data, and also generates process chains. If there are multiple ways to connect services to a suitable process chain, the JobManager decides which way to take. It also tries to generate chains that leverage data locality. This means it provides hints to the internal task scheduler, which in turn executes processing services on those compute nodes that contain the input data. This avoids time-consuming data transfers and reduces required bandwidth. The rule system also puts services accessing the same data into the same process chain, which is then executed on a single compute node. The JobManager's Rule System can therefore be compared to a query optimiser in a relational database (Chaudhuri, 1998), or a multi-query optimiser trying to identify commonalities between multiple queries and to reuse resources (Roy & Sudarshan, 2009).

Rule-based systems have been used successfully for database query optimisation (Pirahesh, Hellerstein, & Hasan, 1992). Hong, Riedewald, Koch, Gehrke, & Demers (2009) have applied a rule-based approach to multi-query optimisation and achieved good results although they propose to combine it with a cost-based approach. This has been tried for single queries by Warshaw & Miranker (1999) who claim to have achieved much better results than earlier work. Some database management systems such as Oracle have deprecated their rule-based optimiser in favour of a more comprehensive cost model (Ahmed et al., 2006). Query optimisation based on costs often leads to faster queries and is more generic than the rule-based approach, which can only cover a finite number of cases. The aim of rule-based approaches, on the other hand, is typically to create an extensible system. Rules are also easier to implement and maintain than a complex cost model.

Our JobManager is not a database management system so the requirements are different. The way services are orchestrated in our work depends on execution costs only to a certain degree. The main goal is to convert a high-level workflow to executable process chains. According to the requirements defined in Chapter 2, *Architecture*, this should be done in an extensible way. The Rule System in the JobManager is configurable, which allows it to be adapted to different application domains as well as to various executing infrastructures. Since execution times of processing services vary to a large degree depending on the input data and are typically unknown in advance, we do not employ a cost model.

3.4 JobManager architecture overview

The main purpose of the JobManager is to control the processing of geospatial data by executing processing services (see Section 2.6) on virtual machines in the Cloud. Figure 3.1 gives an overview of the JobManager's software architecture.

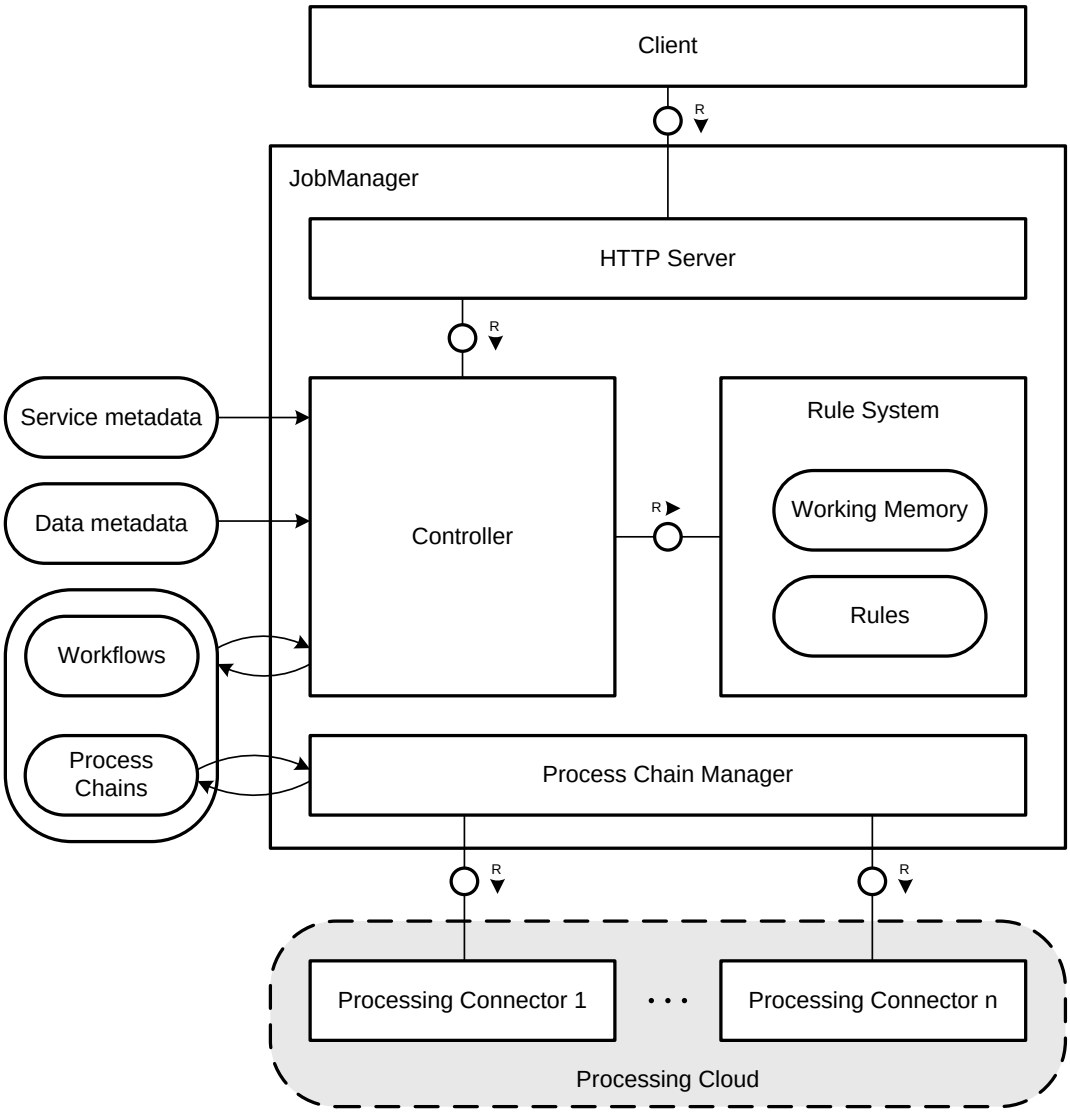


Figure 3.1 Software architecture of the JobManager

The architecture consists of the following components:

- The *HTTP Server* is the main entry point to the JobManager (see Section 3.7.1).
- The *Controller* processes workflows and oversees their execution. It loads metadata about processing services as well as the data to be processed, and calls the *Rule System* to generate executable process chains (see Section 3.7.2).
- The *Rule System* is responsible for generating process chains from workflows, service metadata and data metadata. These process chains are actual instructions telling the *Process Chain Manager* how to execute processing services in the Cloud (see Section 3.7.3).
- The *Process Chain Manager* executes process chains in the Cloud and monitors their execution (see Section 3.7.4).

In addition, there is another component that does not directly belong to the JobManager. The *Processing Connector* is a microservice running on the individual compute nodes in the Cloud. It receives requests from the *Process Chain Manager* and executes the processing services locally on its respective compute node (see Section 3.7.5).

In order to generate the process chains, the *Rule System* makes use of metadata about services and the data to be processed. The *service metadata* contains information about how to execute processing services. This includes a description of their parameters, cardinalities, supported input and output types, etc. (see Section 3.6.2). The *data metadata* describes the datasets stored in the Cloud, in particular properties such as accuracy, acquisition date and owner (see Section 3.6.3).

The JobManager keeps workflows and process chains currently being executed in a database. This is necessary in order to *a)* store results of workflows which can later be queried through the *HTTP Server* and *b)* keep track of running workflows so they can be resumed or restarted in case of a failure (see Section 3.8). This database is also the main communication channel between the *Controller* and the *Process Chain Manager* (see Section 3.7.2).

We implemented the JobManager in Java using the Vert.x tool-kit (Eclipse, 2017) which allows for creating reactive applications. According to the Reactive Manifesto such an application is responsive, resilient, elastic, and message-driven (Bonér, Farley, Kuhn, & Thompson, 2014).

- **Responsive.** All operations in the JobManager happen asynchronously and non-blocking. The software is therefore always able to respond to user requests quickly.
- **Resilient.** The JobManager is resilient to failures and continues to operate in most cases—e.g. if a workflow execution has crashed or if a compute node was unavailable (see Section 3.8).
- **Elastic.** Since the JobManager runs in the Cloud it can react to varying workload. The software itself can be deployed redundantly and scaled up easily due to the clustering features of Vert.x. In addition, the number of compute nodes the JobManager distributes workflow tasks to can be increased dynamically (see Section 3.10)
- **Message-driven.** Each JobManager component is implemented as a Vert.x verticle. Verticles are isolated from each other and operate independently. They communicate through messages that they send asynchronously over the Vert.x event-bus.

3.5 Overview of workflow execution

Figure 3.2 depicts how a workflow is processed by the system—from its definition in the workflow editor to the actual execution of processing services. Note that the figure does not show the archi-

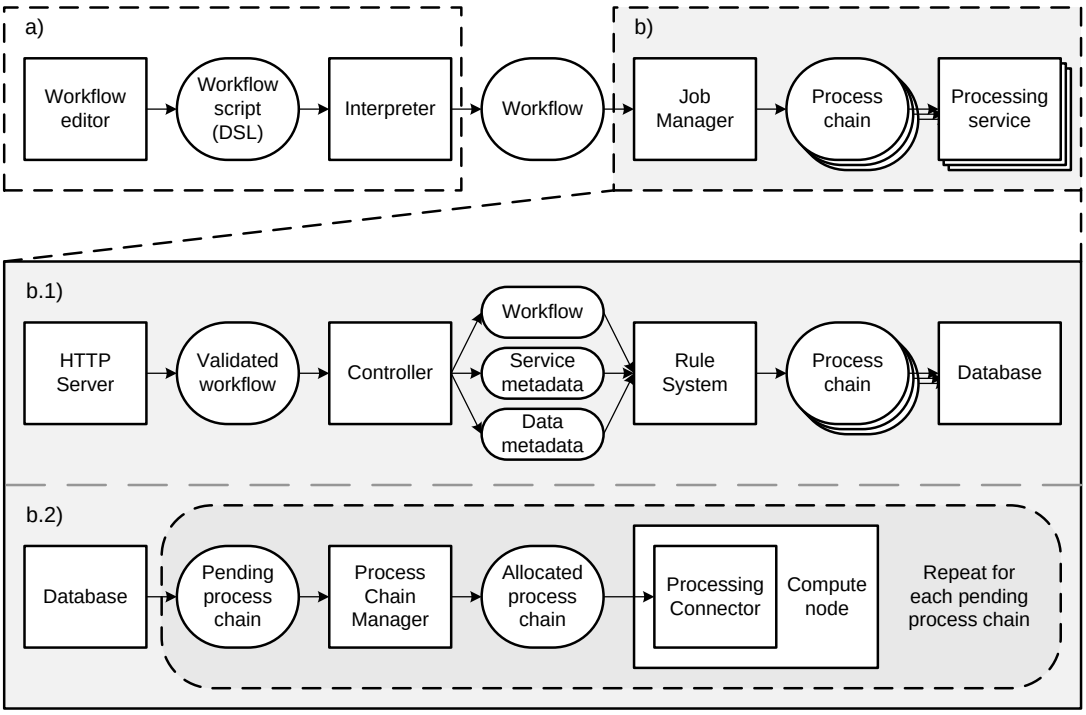


Figure 3.2 Flow of a workflow through the system from its definition to the actual execution

tecture of the JobManager but the conceptual flow of a workflow through the system. It follows up on the overview of workflow execution in Section 2.10 and shows which roles the individual components and data models play in the workflow execution.

The overall workflow execution is divided into two steps. In step a) the human-readable workflow definition is translated to a machine-readable representation. This process is described in detail in Chapter 4, *Workflow Modelling*. Step b) is further divided into two sub-steps. First, the workflow is transformed to one or more executable process chains in step b.1). After this, in step b.2), each process chain is allocated to a compute node in the Cloud and executed.

Step b.1). The *HTTP Server* receives the machine-readable workflow (see Section 3.6.1). It validates the workflow and registers it in a database (see Section 3.7.1). After this, it sends the workflow to the *Controller* which is the main component responsible for the workflow execution (see Section 3.7.2). The *Controller* calls the *Rule System* and forwards the workflow as well as information about the processing services (*service metadata*, Section 3.6.2) and data sets (*data metadata*, Section 3.6.3). The *Rule System* transforms the workflow to one or more *process chains* which are executable descriptions of calls to processing services (see Section 3.6.4). It uses the *service metadata* to decide which processing services need to be called and what parameters need to be provided to them. The *data metadata* is used to decide which datasets should be processed. The resulting process chains are stored in a database.

Step b.2). The *Process Chain Manager* regularly polls this database and looks for *process chains* that have not been executed yet. The *Process Chain Manager* is responsible for scheduling their execution (see Section 3.7.4). This means it allocates each *process chain* to an available compute node and sends it to an instance of the *Processing Connector* running on this node. The *Processing Connector* finally calls the processing services and collects their output (see Section 3.7.5).

In the following sections each data model and component contributing to step b) is described in detail.

3.6 Data models

In this section we specify the data models of our JobManager. We first introduce the model for workflows. Then we specify which information must be provided about processing services so they can be executed by the JobManager (service metadata). We also describe the metadata that helps our system select the data to be processed (data metadata). Finally, we specify the data model for executable process chains which are generated by the Rule System based on the workflow model, the service metadata and the data metadata.

3.6.1 Workflow model

Figure 3.3 depicts the data model for workflows executable by the JobManager. On submission, each workflow has a unique identifier, a name, a list of variables, and a list of actions. When the workflow is executed, the JobManager sets additional attributes denoting the time the execution has started, the current processing status, and finally the time the execution has finished.

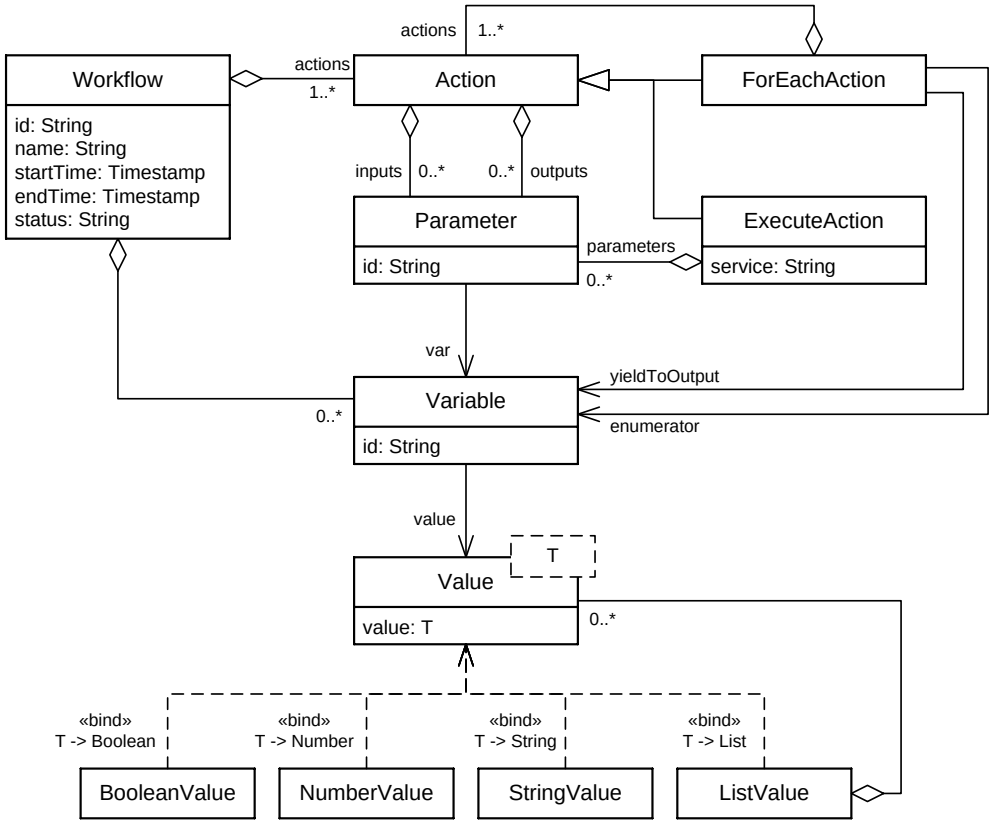


Figure 3.3 Class diagram of the workflow model

The list of variables contains the workflow input parameters and declarations for variables used within the workflow. Each variable is a key-value pair, whereas the key is an identifier and the value is either a primitive boolean, number or string, or a list of primitive values. For pure variable declarations, the value can be undefined. Variable declarations are typically used to create links between the outputs and inputs of actions. Table 3.1 lists examples for variables and their meaning.

Example ID	Example value	Description
input0	/input-data/point-cloud.las	A path to an input file
output0	/home/user/workflow134/terrain.tif	A path to an output file
resolution	50	An input argument
input1	[/input-data/point-cloud-01.las, /input-data/point-cloud-02.las]	A list of input files
enum0		A declaration of a variable that can be used as an enumerator in a for loop
var0		A declaration of a variable that can be used as an output parameter of a service and an input parameter of another one

Table 3.1 Examples for workflow variables

The workflow also contains one or more actions to execute. There are two types of actions: executable actions and for-each actions. Each action has input and output parameters. Executable actions refer to a processing service by its name and have additional parameters that should be passed by value to the service during execution. Parameter identifiers match the ones specified in the service metadata (see Section 3.6.2).

For-each actions have a list of sub-actions that should be applied to a list of input data. Each for-each action also has an enumerator which is a variable the sub-actions can use to refer to an item in the input list. A for-action also makes use of another variable declaration which can be used to collect (or yield) output from sub-actions to an output list.

The workflow status is a string that is set by the JobManager during execution. Table 3.2 shows all possible values.

Status	Description
`ACCEPTED`	The workflow was accepted by the JobManager's HTTP interface and scheduled for execution
`RUNNING`	The workflow is currently being executed
`SUCCESS`	The workflow execution finished successfully
`ERROR`	The workflow could not be executed
`PARTIAL_SUCCESS`	The workflow was executed, but some input datasets could not be processed successfully

Table 3.2 Possible workflow statuses

3.6.2 Service metadata

One of the main benefits of our approach is the ability to orchestrate existing processing services without requiring their developers to implement a special interface. For this purpose, our system relies on lightweight service descriptions stored in JSON files. We call these files *service metadata*. They contain information about the services (such as IDs, names, or descriptions) and describe input and output parameters.

Note that we define the service metadata format here and not a specific interface for the processing services. The service metadata format actually is a generic way to describe a service interface. It is designed to not impose any additional requirements on the services other than the ones described in Section 2.6.

Table 3.3 lists all service metadata attributes for the description of a single processing service. Except for *runtime_args*, all attributes are mandatory.

Key	**Type**	**Description**	**Example**
id	string	Unique service identifier	35
name	string	Human-readable name	ResamplingOfPointCloud
description	string	Human-readable description	Resamples a point cloud
type	string	Service type	Resampling
path	string	Relative path to the service executable in the service artefact	bin/resample.sh
os	string	Operating system	linux
runtime	string	Runtime environment (see Table 3.4)	other
runtime_args *(optional)*	string	Additional arguments to pass to the service runtime	
parameters	array	Array of parameters (see Table 3.5)	[{ … }, { … }, …]

Table 3.3 Service metadata: description of attributes

The attributes *name* and *description* should contain human-readable strings. They are used in our system's main user interface and in the Workflow Editor described in Chapter 4, *Workflow Modelling*. The editor can be used by domain experts to define their own workflows in a Domain-Specific Language (DSL). The editor's window contains a cue card with information about the language keywords, as well as the names and descriptions of processing services the domain experts can use in their workflows. The DSL has a generic *apply* keyword allowing domain experts to directly call a processing service by its name (see Section 4.5.5). Since service names are represented by identifiers in the Domain-Specific Language they must not contain whitespaces (see Appendix A, *Combined DSL grammar*). For the sake of consistency, all service names should be in camel case and begin with a capital letter. Valid examples are *ResamplingOfPointCloud* or *SplineInterpolation*. The identifier *My cool new service* is invalid.

The attribute 'type' is a string defining a specific type of processing service. This attribute can be used by the Rule System to identify services that perform the same type of operation. The interpretation of this parameter depends on the environment the system is used in and on the registered processing services. Since there are so many different types we do not specify a certain list here. Example values are, however, 'Interpolation' or 'Co-registration'.

The metadata attribute 'path' specifies the relative path to the service executable in the artefact or archive (e.g. ZIP file) containing the processing service.

The attribute 'os' specifies the operating system the processing service requires. This information is used by the Rule System and the Process Chain Manager to distribute services to compatible nodes in the Cloud. Our implementation currently only supports a value of 'linux', but other possible values are 'windows' or 'macos'.

The service metadata attribute 'runtime' is used by the JobManager to decide which environment the processing service must run in. Table 3.4 contains all possible values. If the attribute equals 'spark', for example, the JobManager will treat the processing service as a Spark applica-

tion and submit it to a compute node running a Spark cluster manager. The most generic value is 'other' which means the service is an arbitrary executable program. In combination with the ability to run services inside Docker containers, our system in fact supports arbitrary programs.

Value	Description
shell	Service is a bash script (Bourne-Again SHell)
java	Service will be executed in a Java Virtual Machine (JVM)
python	Service is a python program
mono	Service will be executed in the Mono runtime
hadoop	Service is an Apache Hadoop MapReduce job in a jar file
spark	Service is an Apache Spark application in a jar file
other	Service is an arbitrary executable program

Table 3.4 Service metadata: runtime environments (possible values for the 'runtime' attribute)

In addition, the optional attribute 'runtime_args' can be used to pass arguments to the service runtime. For example, if the service is a Java program, 'runtime_args' may be `-Xmx512M` specifying its maximum heap size.

The service metadata also contains a description of all processing service parameters. The attribute 'parameters' is an array of JSON objects. Table 3.5 describes their structure.

Key	Type	Description	Example
id	string	Unique parameter identifier	output_file_name
name	string	Human-readable name	outputFileName
description	string	Human-readable description	Output file
type	string	Parameter type (either 'input', 'output', or 'argument')	output
cardinality	string	Parameter cardinality ('min..max')	1..1
default *(optional)*	any	Default value for this parameter	
data_type	string	Parameter type (see Table 3.6)	Point Cloud
file_suffix	string	Output file extension or suffix	.las
label *(optional)*	string	Parameter name on the command line	`--output`
dependencies *(optional)*	array	Parameters this parameter depends on	
conflicts *(optional)*	array	Parameters this parameter conflicts with	

Table 3.5 Service metadata: description of parameters

The attribute 'id' must be a string that identifies the parameter uniquely within the service metadata. A parameter description also contains a human-readable 'name' and 'description'. Just

like the service name and description, these attributes are displayed in the Workflow Editor. The parameter name can become part of the Domain-Specific Language and must therefore not contain whitespace characters. It should be in camel case and start with a lower-case letter, so it can be differentiated from the service name.

The attribute 'cardinality' defines the minimum and maximum number of occurrences of a parameter in a command line. If the minimum is `0` the parameter is optional. If it is at least `1` it is mandatory. The maximum must never be lower than the minimum. The character `n` means an indefinite number. Valid values for this parameter are, for example, `1..1` (the parameter must be provided exactly once), `0..1` (the parameter is optional but can be provided once), `1..4` (the parameter is mandatory and may be provided up to four times), `0..n` (the parameter is optional and may be provided an indefinite number of times).

Note that the optional attribute 'default' can be used to specify a default value for mandatory parameters. This is useful if a processing service requires a specific parameter but it should be hidden from the user in the workflow description. Mandatory parameters with a default value become optional in the Domain-Specific Language.

The parameter description also contains an attribute named 'data_type' that is used by the Rule System to determine if the output parameters of a service are compatible to the input parameters of a subsequent one. The attribute specifies the parameter type in terms of what kind of data it accepts. The value can either be one of the primitive types defined in Table 3.6 or an arbitrary string. In the latter case, the parameter refers to either an input or output file and denotes the type of the dataset in this file. Valid strings relevant to this thesis are, for example, `Point Cloud` or `Triangulation`.

Value	**Description**
integer	The parameter is an integer value.
float	The parameter is a floating point number.
string	The parameter is a string.
boolean	The parameter is a boolean value. Valid values are `true`, `1`, `yes` and `false`, `0`, `no`.
directory	The parameter is a string specifying a directory in the distributed file system containing input files.
<other>	An arbitrary string denoting the type of the dataset in an input or output file

Table 3.6 Service metadata: parameter types (possible values for the 'data_type' attribute)

The attribute 'file_suffix' is only valid if the parameter 'type' is `output`. In this case it specifies a string that the Rule System will append to all generated output file names. This attribute is typically used to specify a file extension such as `.tif` or `.las`.

Parameters on the command line are often differentiated from each other by strings starting with a slash, a dash or a double-dash. The attribute 'label' can be used to specify such a string. Valid values are, for example, `--input`, `--output`, `-i`, `-o`, or `/tolerance`.

Many programs accept parameters without labels. Instead, they rely on the order in which the parameters are given on the command line. As mentioned above, we do not require the processing services to implement a specific interface. In our parameter description the attribute 'label' is therefore optional. The JobManager produces service calls with parameters that are in the same order as the descriptions in the service metadata attribute 'parameters'.

The attributes 'dependencies' and 'conflicts' are optional arrays of parameter identifiers. They can be used to specify dependencies between parameters—e.g. if a certain parameter requires another one to be given as well—or conflicts—e.g. if two or more parameters are mutually exclusive.

3.6.3 Data metadata

In order to be able to automatically select datasets to be processed from the distributed file system, the JobManager makes use of *data metadata*. This metadata contains information about the datasets such as their extent or spatial resolution.

ISO 19115-1 (2014) specifies a complex schema with a large number of attributes describing geographic information. This standard is well known in the geospatial community and widely adopted amongst data providers and users. The JobManager is able to process ISO 19115-1 metadata and to use it in the Rule System to decide which datasets to process. The following is an excerpt of ISO 19115-1 attributes giving examples how they can be utilised.

Attribute:	`MD_DataIdentification.extent`
Description:	The spatial extent of a dataset
Example:	Find datasets within a certain area.

Attribute:	`MD_DataIdentification.spatialRepresentationType`
Description:	The type of information stored in the dataset
Example:	Select datasets containing grids, vector data, etc.

Attribute:	`MD_Metadata.dateStamp`
Description:	The date and time the dataset was created
Example:	Compare two datasets and select the one that is newer.

Attribute:	`MD_DataIdentification.spatialResolution`
Description:	The density of a dataset
Example:	Compare the spatial resolution of two datasets and find a trade-off between required quality and processing time. The dataset with the higher resolution might have a better quality, but the one with the lower resolution might be faster to process.

Attribute:	`MD_DataIdentification.pointOfContact`
Description:	The party responsible for the dataset
Example:	A dataset that is provided or maintained by a certain party may be more reliable because the party has a higher credibility.

3.6.4 Process chain model

The process chain model describes a set of actions that are executed on the compute nodes in the Cloud as well as any dependencies between them. Process chains are created by the Rule System. The Process Chain Manager oversees the execution of process chains, and the Processing Connector uses them to create command lines for the processing services.

Figure 3.4 shows the class diagram of the process chain model. A process chain contains a set of executables that represent calls to processing services. An executable has a path pointing to the

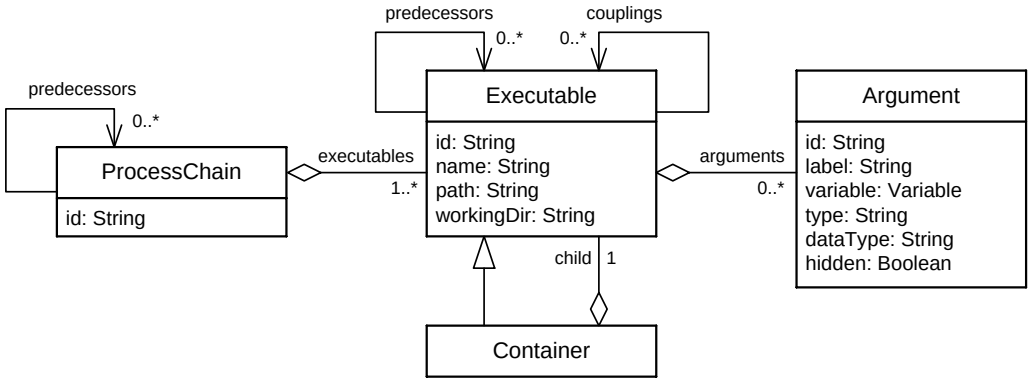


Figure 3.4 Class diagram of the process chain model

actual processing service executable (see the service metadata attribute 'path' in Section 3.6.2) as well as arguments that should be passed to the service. The arguments have labels, types and data types that correspond to the attributes from the service metadata. In addition, an argument refers to a variable containing the argument's value.

Since processing services are typically run in parallel, the set of executables in a process chain is not necessarily an ordered sequential list. Instead, an executable can have one or more predecessors specifying other executables in the same process chain that must be executed before this one.

Some attributes of the process chain model are needed by the Rule System and some are necessary for the Process Chain Manager and the Processing Connector. For example, process chains may have predecessors just like the executables. The Rule System uses this information to build a directed graph of process chains from a workflow. It keeps this information internal and only returns process chains to the Controller that do not have predecessors or those whose predecessors have already been executed successfully.

The same applies to containers. A container represents a runtime environment such as Docker or the Java Virtual Machine (see service metadata attribute 'runtime' Section 3.6.2). They wrap around executables and specify that a certain container application should be executed instead of the processing service itself. This information is needed by the Rule System to correctly generate calls to processing services.

In addition, the Rule System specifies couplings between executables that should be run within the same process chain and on the same compute node. This information is only necessary during the creation of process chains and will be ignored later on.

Arguments can be hidden, which means the Processing Connector will ignore them when it creates the command lines for the processing services. Instead, such arguments may be used by the Rule System internally to link executables together.

Finally, the working directory of an executable depends on the compute node where the process chain will be run. It will be set by the Processing Connector and ignored by the Rule System.

Note that multiple process chains may be created for a single workflow, depending on how much information is available at the time of creation and how the services should be distributed to the compute nodes. For more information, see Section 3.7.3.

3.7 Components

In this section we describe the active components that participate in the processing of geospatial data in the Cloud. We give details on the individual parts of the *JobManager*, as well as the *Processing Connector* and the processing services.

3.7.1 HTTP Server

The *HTTP Server* is the main entry point to the JobManager. It receives incoming HTTP requests from clients such as our system's main user interface. Figure 3.5 depicts the control flow when a client sends a workflow to the JobManager.

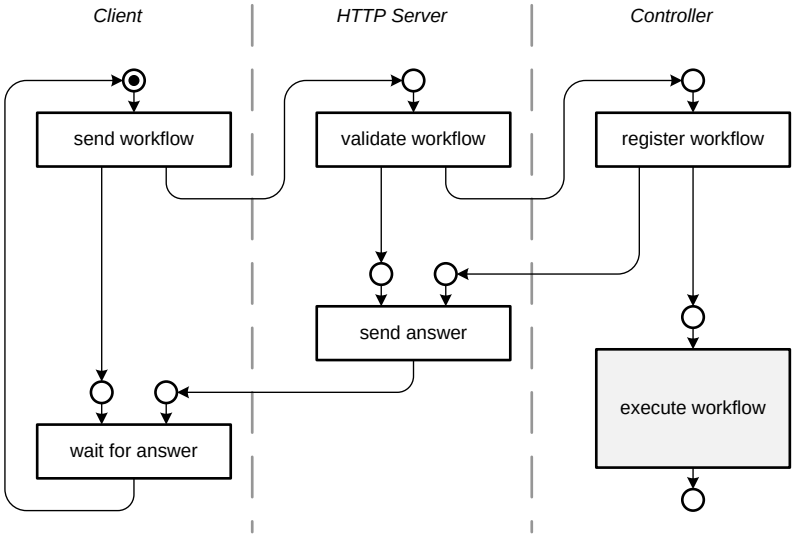


Figure 3.5 Control flow in the JobManager when the HTTP Server receives an AWF from a client

The flow starts in the upper left corner. The *Client* sends a workflow to the *HTTP Server* and then waits for an answer. Depending on the client's implementation, this most likely happens asynchronously. The *HTTP Server* validates the workflow in terms of syntax and semantics. It immediately sends an error back to the *Client* if the workflow is invalid. If it is valid, the *HTTP Server* asynchronously sends the workflow to the *Controller* which generates a unique identifier for it and then registers the workflow in its database. Before the *Controller* executes the workflow, it sends the unique identifier back to the *HTTP Server* which in turn forwards it to the *Client* with an HTTP status code 202 (Accepted). This specific code means that the workflow was accepted and that it is now scheduled for execution, but the actual execution will happen at a later point in time—unless something prevents the workflow from being executed (see status code description in Fielding & Reschke, 2014, p. 52). It is important that the *Controller* saves the workflow to a persistent database before it sends its answer. This ensures the workflow can be executed and the JobManager's answer will be valid even if there is a system failure before the execution starts (see Section 3.8).

When the *HTTP Server* has sent the answer to the client the *Controller* will finally launch the workflow. This process is described in Section 3.7.2. While the workflow is being processed, the client may regularly poll its status by sending a request to the JobManager using the unique identifier it received earlier. In such a case, the *HTTP Server* queries the *Controller* to lookup the workflow's status in its database.

The following is a list of endpoints and operations the *HTTP Server* supports.

POST workflow — Endpoint: /

This endpoint can be used to send a workflow to the JobManager. The *HTTP Server* will schedule it for execution and return a unique identifier with which the workflow's status can be queried.

Parameter	Description
body	Workflow that should be processed.

Status code	Description	Response body
202	The workflow was accepted and is now scheduled for execution.	A string that uniquely identifies the accepted workflow.
400	The provided workflow was invalid.	None.

GET list of workflows — Endpoint: /workflows

With this endpoint, a client can get information about all workflows registered in the JobManager's database including their status.

Parameter	Description
offset	The index of the first workflow to return, starting with zero. *(optional, default: 0)*
limit	The maximum number of workflows to return. *(optional)*
sort	The name of a property from the workflow model (see Section 3.6.1) used to sort the returned list. Valid values are "id", "name", "startTime", "endTime", and "status" *(optional, default: "startTime")*
order	A positive number if the returned list should be sorted ascending, a negative number if it should be sorted descending. *(optional, default: -1)*

Status code	Description	Response body
200	The workflow list was generated successfully.	A JSON array containing all workflows according to the model defined in Section 3.6.1.
400	One of the given parameters was invalid.	None.

GET workflow details — Endpoint: /workflows/:id

This endpoint can be used to get information about a workflow registered in the JobManager's database including its status.

Parameter	Description
id	The unique identifier returned by the HTTP Server when the workflow was submitted.

Status code	Description	Response body
200	The workflow was retrieved successfully from the database.	A JSON object representing the workflow according to the model defined in Section 3.6.1.
404	The requested workflow was not found in the database.	None.

DELETE workflow — Endpoint: /workflows/:id

With this endpoint, a workflow can be removed from the Controller's database and the execution can be cancelled. This operation also deletes all generated process chains belonging to the workflow.

Parameter	Description
id	The unique identifier returned by the HTTP Server when the workflow was submitted.

Status code	Description	Response body
204	The workflow was deleted from the database or it did not exist in the first place (the operation is idempotent).	None.

3.7.2 Controller

The *Controller* is one of the main components of the JobManager. It receives workflows from the HTTP Server, calls the Rule System (see Section 3.7.3) to generate process chains for these workflows, and stores them in a database so they can be executed by the Process Chain Manager (see Section 3.7.4). It also regularly looks up process chain results to update the status of workflows currently being executed.

The database maintained by the Controller contains received workflows and generated process chains. It serves three purposes:

- The stored workflows can be returned to the client on request (see Section 3.7.1).
- Since the database is a persistent representation of the JobManager's state, it can be used to continue or restart a workflow execution in case of a system failure without losing information (see Section 3.8).
- The database is used as the main communication channel between the Controller and the Process Chain Manager. Both components do not communicate with each other directly but alter the contents of the database and regularly look for updates.

Figure 3.6 depicts the control flow in the JobManager when the Controller has received a workflow from the HTTP Server and monitors its execution. It extends Figure 3.5 in which it represents the grey transition box labelled "execute workflow".

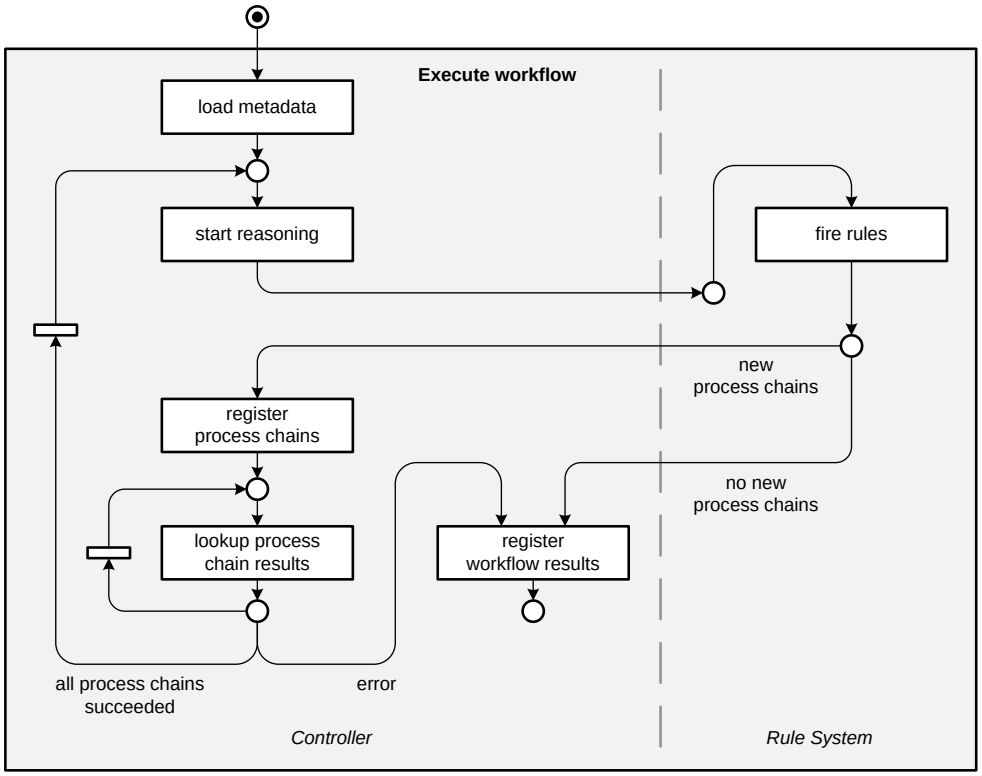


Figure 3.6 The control flow in the JobManager while the Controller executes a workflow

First, the Controller loads the service metadata and the data metadata. It then calls the Rule System and forwards metadata as well as the workflow. The Rule System puts this information in its working memory and fires all rules until it has either generated new process chains or until there are no further rules to be fired.

If the Rule System has created new process chains, the Controller stores them in its database. As we show in Section 3.7.4, the Process Chain Manager picks up the process chains from the database and executes them in the Cloud. In the meantime, the Controller regularly polls the database and checks if there are new process chain results added by the Process Chain Manager. If the Controller finds a failed process chain in the database it will abort the execution of the whole workflow. In this case, it will register the results of successful process chains belonging to the same workflow in the database and set the workflow's status to `PARTIAL_SUCCESS`. If the workflow does not have any successful process chains, the Controller will set the status to `ERROR`.

If the Controller has polled the database and all process chains of the workflow have succeeded, the Controller calls the Rule System again to see if it generates any more process chains. This time the Controller also forwards the results from all previous process chains of the workflow currently being executed to the Rule System (in particular their statuses and the created output files) so it can reason about them as well.

The whole process continues until the Rule System finishes without generating more process chains. In this case the workflow is considered finished since there are no further process chains to be executed. The Controller registers the workflow results in the database, sets the workflow's status to `SUCCEEDED`, and stops the execution.

Note that since the Controller is implemented in an event-based, asynchronous way it can monitor the execution of multiple workflows at the same time.

3.7.3 Rule System

The Rule System is responsible for creating executable process chains from workflows. It consists of two parts: a working memory and a set of production rules. The working memory contains the facts the rule system can reason about. In our case these are the data metadata, the service metadata and the workflow to execute. In addition, the Controller inserts artificial facts for successfully executed process chains. This is necessary so that the Rule System can generate and return more process chains (see the paragraph named "Result" below).

Production rules typically consist of a left-hand side and a right-hand side containing conditions and actions respectively. The Rule System evaluates the facts in the working memory against the conditions. If all conditions of a rule become true it will "fire", which means the actions will be executed. Rules that are firing may change the contents of the working memory. Other conditions may then become true and hence other rules will fire. This way, rules can be connected to a network performing complex operations. In our case we use production rules to quickly reason about data metadata, service metadata and workflow actions. Depending on this information we prepare optimized process chains that can be executed in a distributed way in the Cloud by the Process Chain Manager. Using a production rule system also allows us to keep this process configurable and adaptable to different use cases and scenarios.

The production rules are divided into three phases that are executed sequentially. First, the Rule System selects the datasets to process, then finds processing services, and finally generates the process chains.

Select datasets

In Chapter 4, *Workflow Modelling* we present our Domain-Specific Language for workflow modelling. This language allows users to either specify a dataset to process directly by its filename on the distributed file system (using placeholders; see Section 4.5.5) or to just specify a data type such as 'PointCloud'. In addition, the keyword 'recent' allows users to select the most up-to-date dataset from a set of candidates.

These language features can be mapped to production rules. For example, the workflow expression 'latest PointCloud' can be implemented by creating two rules: one that reasons about the data metadata and creates a set of datasets with the type 'PointCloud', and another one that selects the most up-to-date dataset from this set.

Further rules can be implemented, for example, to select datasets by their spatial extent, by a specific owner, or based on the permissions a user has (see Section 3.6.3 for more examples). As mentioned above, the Rule System is a dynamic and configurable part of our system that allows us to adapt the behaviour to various requirements.

Select services

The Rule System also selects the processing services that are applied to the chosen datasets in a workflow. For this, it reasons about the service metadata and the individual actions in the workflow as follows:

1. The first production rule creates a set of matching candidates for each workflow action based on the service name, the parameters, parameter types, cardinalities, etc.

2. The Rule System also considers subsequent services and checks if their input and output parameters are compatible to each other in terms of data types. This further reduces the set of candidates.

3. Another production rule selects the service with the highest semantic version from the remaining set of candidates.

4. If the set is empty, the Rule System aborts the workflow execution, as the workflow is apparently invalid. Otherwise, a final rule converts the selected service candidate to an `Executable` instance from the process chain model (see Section 3.6.4).

Again, the Rule System allows us to keep our architecture configurable. For example, if for a certain use case, it is necessary to always select a service with a specific version number (e.g. because it is the most reliable one known to work well with the input datasets from this use case), or if a certain group of users has licenses for some processing services but not for others, this can be configured in the Rule System.

Generate process chains

The production rules selecting the processing services convert the service candidates to `Executable` instances from the process chain model (see Section 3.6.4). Next, these `Executable`s need to be linked and put into process chains, which will be executed by the Process Chain Manager in the Cloud. All `Executable`s in a process chain are executed on the same compute node, but multiple process chains can run in parallel in the Cloud. Process chains are created as follows:

1. First, for-each actions are *unrolled*, which means they are replaced by copies of their sub-actions whereas the number of copies depends on the list of inputs the for-each action is applied to. For example, consider a for-each action F with the sub-actions A and B. If F should be applied to five inputs, A and B will be copied five times and F will be removed from the working memory. The inputs of A and B will be adapted accordingly.

2. The parameters of workflow actions are converted to arguments for the respective `Executable`s. This includes adding default values for optional parameters, appending file suffixes, etc.

3. If required, the individual `Executable`s are linked to further processing services that should be executed either before or after the one the `Executable` refers to. This includes adding services for data conversion, monitoring, or other purposes.

4. `Executable`s referring to processing services that require a special environment are wrapped into a `Container`. This allows the Rule System and the Processing Connector to generate correct command lines.

5. `Executable`s are finally connected to a directed graph by comparing the `Variable` instances of their input and output arguments. An `Executable` with an output argument that equals an input argument of another one becomes a predecessor of this `Executable`.

The graph of executables is then converted to process chains according to the following constraints:

- Create a new process chain for every `Executable` that does not have a predecessor.
- Keep couples (i.e. processing services that need to be executed on the same compute node) together.
- Try to optimize the process chains and leverage data locality by putting `Executable`s that refer to the same datasets in the same process chain.
- Try to leverage parallelisation as much as possible by splitting process chains at junction points and putting the individual paths in separate process chains. Keep the dependencies between the `Executable`s by correctly setting process chain predecessors.

The last point is very important for our system as it allows us to create independent process chains that can be executed in parallel on multiple compute nodes. Figure 3.7 illustrates an example of a process chain and how it is split into multiple ones. `Executable` 1 has three outputs that should be processed in parallel by `Executable`s 2 to 7. `Executable` 8 merges the outputs together. The process chain can be split into five smaller ones. Process chains 2 to 4 each contain two `Executable`s. They can be run in parallel in the Cloud because they do not have dependencies to each other. All `Executable`s in a process chain are run on the same compute node to leverage data locality. The execution of process chain 5 can only commence once chains 2 to 4 are finished. This is ensured by the predecessor dependencies between the process chains.

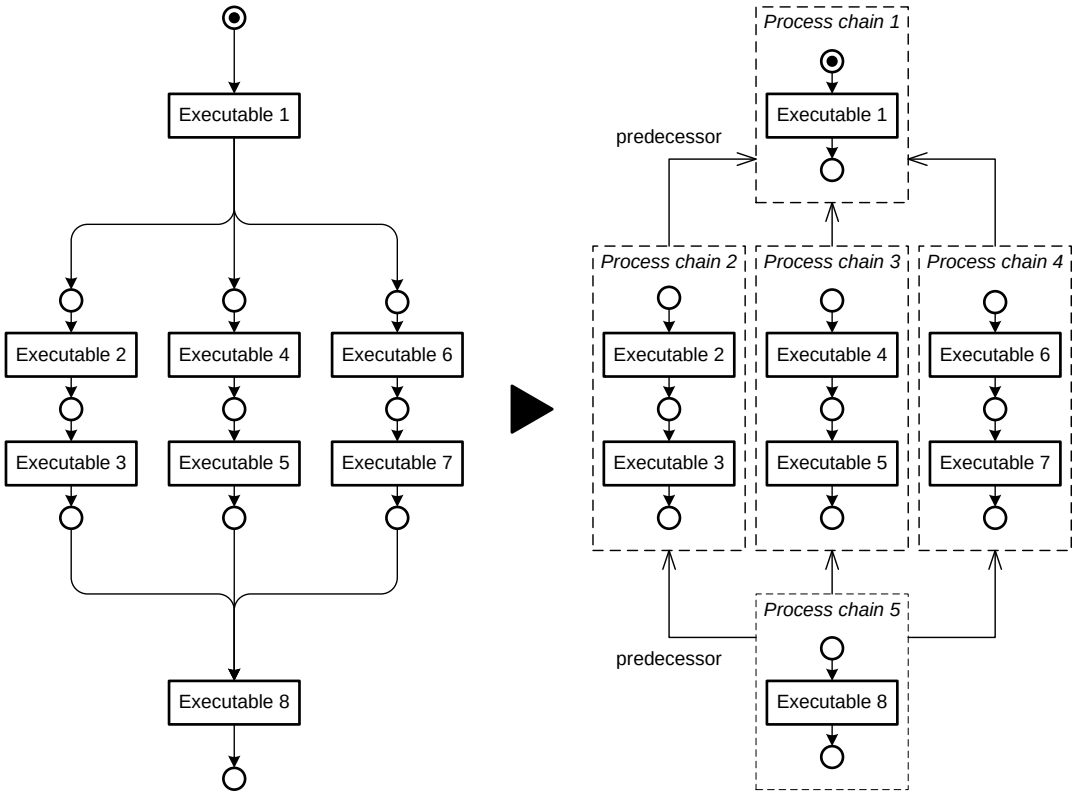


Figure 3.7 A process chain that is split into multiple ones to leverage parallelisation

Data locality optimisation

As described above, each process chain is executed on exactly one compute node. This keeps services that access the same data together. For example, a service that writes a certain file and a subsequent one reading this file are put into the same chain—as long as the constraints from the previous subsection are met. This avoids data unnecessarily being transferred from one compute node to another, which would increase network usage and possibly slow down workflow execution.

Another optimisation that the JobManager performs is based on the location of input files from the original data set the workflow is applied to. Before the workflow is executed, the JobManager creates an index of all files that will be processed and determines their location in the Cloud. It puts this information in the working memory of the Rule System. Based on this, the Rule System creates hints for the Process Chain Manager (see Section 3.7.4) telling it on which node a generated process chain should be executed. Again, this reduces the need to transfer large amounts of data. Instead, the processing services are transferred to (or executed on) the compute nodes that contain the requested files. Data locality optimisation is one of the approaches that help the JobManager to make best use of available Cloud resources.

As described in Chapter 2, *Architecture* we use a distributed file system for data storage. Since such a file system supports replication and several copies of an input file may be located on multiple compute nodes, the hints that the Rule System forwards to the Process Chain Manager may contain several locations. The Process Chain Manager is optimised for high throughput. Based on the hints it will select an available compute node to execute the process chain on. If none of the available nodes matches the locations in the hint, it will either delay the process chain and continue with others or execute it on any available node—whichever option is faster.

Result

At the end of the reasoning process the Rule System returns the process chains that are ready for execution—i.e. those without a predecessor or whose predecessors have already been executed successfully.

As mentioned above, whenever a process chain was executed successfully, the Controller adds an artificial fact into the Rule System's working memory including the process chain's status and its outputs—i.e. the files created by the processing services. This allows the Rule System to apply for-each actions to these outputs and to generate the correct number of process chains. The ability to produce process chains (or workflow branches) dynamically at runtime without a priori knowledge at design time corresponds to workflow control-flow pattern WCP-14 (see Section 3.2.2). This differentiates our approach from many other workflow management systems for distributed computing.

3.7.4 Process Chain Manager

The Process Chain Manager distributes process chains to compute nodes in the Cloud. It oversees their execution and handles results and failures. Figure 3.8 illustrates the component's main loop.

The Process Chain Manager periodically polls the JobManager's database for process chains. If there is a new chain that has just been generated by the Rule System and that is now ready for execution, the Process Chain Manager selects a compute node to execute it on (according to the algorithm described below). If there is a node available, the Process Chain Manager sends the

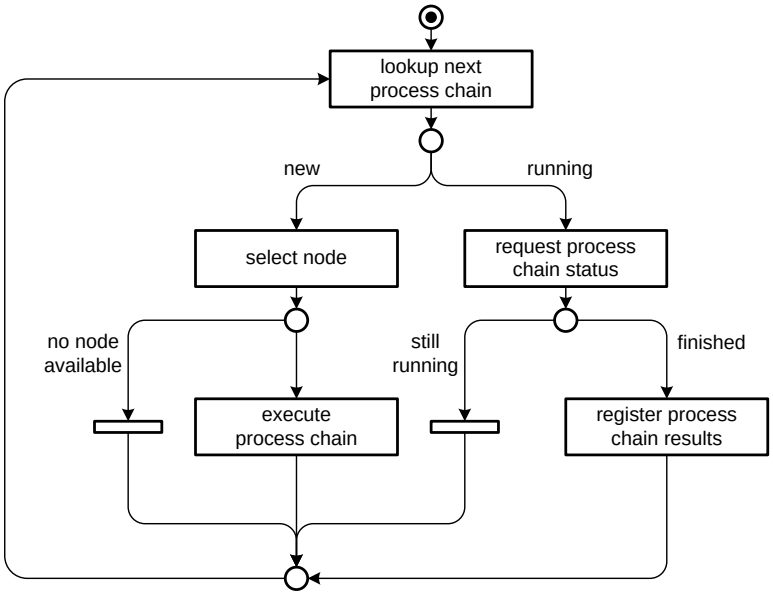


Figure 3.8 The main control-flow in the Process Chain Manager

chain to the Processing Connector running on this node and sets its status to `RUNNING` in the database. Otherwise, it returns to the start and retries the process in the next loop.

Process chains that are currently `RUNNING` are treated differently. The Process Chain Manager first requests the current status from the Processing Connector that runs the process chain. If it is still running, the Process Chain Manager just returns to the start and continues with periodic polling. Otherwise, if the process chain has finished in the meantime, the Process Chain Manager registers its results (i.e. the output files written) in the database and finally returns to the start.

Error handling

The Process Chain Manager handles failures in the following way:

- If the connection to a compute node failed or timed out, or if the Processing Connector returned HTTP status code 500, the Process Chain Manager selects another node and retries the operation after a short delay of up to five seconds.
- If a configurable number of other nodes also fail, the process chain's status is set to `ERROR`. It will not be retried.
- If the Processing Connector successfully returns the result of a process chain but the chain has failed (e.g. because one of the processing services failed with a non-zero exit code, see Section 2.6.1), its status is set to `ERROR` and it is not retried.

Node selection

There are a number of algorithms that can be used to select a compute node on which a process chain should be executed. Russell et al. (2004b) suggest *Random Allocation (workflow resource pattern R-RMA)*, *Round-Robin Allocation (R-RRA)*, or *Shortest Queue (R-SHQ)*. We also consider *First In – First Out (FIFO)* which is a variation of *Shortest Queue*.

- **Random Allocation.** In this algorithm the workflow engine (in our case the Process Chain Manager) keeps a list of resources (i.e. compute nodes) from which it selects one at random.

- **Round-Robin Allocation.** This algorithm aims for an even utilisation of all resources. It selects compute nodes one after the other from the list. If the last node has been reached, the algorithm starts from the beginning.

- **Shortest Queue.** The Process Chain Manager keeps a list of all nodes and a queue of running tasks (in our case process chains) for each node. It selects the node with the shortest queue which is supposed to be the one with the most resources available.

- **First In – First Out (FIFO).** All available compute nodes are kept in a queue. The Process Chain Manager selects the first one and removes it from the queue. Nodes that have finished executing a process chain are put back at the end of the queue. If the queue is empty, the Process Chain Manager has to wait until a node becomes available again.

Shortest Queue is optimised for maximum throughput. We use this algorithm because we want to make use of as much resources as available and to process workflows as fast as possible.

In Chapter 5, *Evaluation* we will see that there are typically a lot more process chains ready to be executed than available compute nodes. In order to prevent the nodes from being overloaded, we limit the size of the queue of process chains per node. The limit can vary amongst the individual compute nodes. A value of 1 basically reduces the algorithm to *First In – First Out*. Since we often deal with processing services that only make use of one CPU core, but the compute nodes offer multiple cores, it is advisable to set the queue size for each node to the respective number of available CPU cores.

Note that the node selection process is not only based on available slots in the queue but also depends on the capabilities offered by a node. The Process Chain Manager analyses the process chains and the service metadata and collects a list of requirements the processing services have towards the environment they are executed in. For example, if the service metadata specifies that a certain service requires Apache Spark the Process Chain Manager will limit the list of compute nodes from which to select to those which have Apache Spark installed.

In addition, as described in Section 3.7.3 the Rule System performs data locality optimisation to reduce the amount of data transferred over the network and to improve workflow execution performance. It generates hints for the Process Chain Manager to tell it on which nodes it should run generated process chains. The Process Chain tries to follow the hints and assign process chains to the given compute nodes as long as this does not prevent high throughput. If all of the given compute nodes are currently busy, the Process Chain Manager will either delay the execution of the process chain and continue with another one or execute it on any available node—depending on which option is faster. This means the hints generated by the Rule System are only a means to prioritise certain compute nodes over others, but not a guarantee that the process chain will actually be executed on these nodes.

Another way of optimising throughput is to use *Random Allocation* or *Round-Robin Allocation* and to keep a queue at the side of the compute node in the Processing Connector. This way the Processing Connector could monitor resource usage on the compute node and decide on its own when to execute the process chain. However, this could lead to free resources being unused on other compute nodes whose queue is already empty. This approach would therefore require a way for the Process Chain Manager to intervene and to deallocate a process chain (whose execution has not started yet) from a node and to reallocate it to another one.

In order to achieve best results, such an algorithm would also require some kind of knowledge about the processing services and their expected resource usage. Otherwise deciding when to start a process chain containing multiple service calls with different resource requirements would be

mere guessing. *Shortest Queue*, on the other hand, is more predictive, simpler to implement, and less error-prone. As we will show in Chapter 5, *Evaluation* it achieves a reasonable throughput.

Note that scheduling algorithms known from the domain of operating systems such as any preemptive algorithm or priority-based ones that consider the amount of time a task may take do not apply here. Our system does not support pausing and resuming of process chains, nor do we have any information about the expected runtime or resource usage of processing services and hence process chains.

We could assign priorities to process chains in order to be able to execute important workflows before other ones. Such a prioritisation could be done in the Rule System or configured by the user in our system's main user interface. Special care would have to be taken in this case to prevent starvation—i.e. process chains that are never allocated to any node because there always is another one with a higher priority. More complex and sophisticated scheduling algorithms are, however, not the focus of this work and therefore beyond its scope.

3.7.5 Processing Connector

On each compute node in the Cloud there is a Processing Connector instance which receives process chains from the Process Chain Manager through HTTP. It takes care of executing processing services according to the order determined by the predecessor dependencies in the received process chains. For each process chain, it also collects the service outputs and provides them through its HTTP interface to the Process Chain Manager. If one of the outputs is a directory the Processing Connector will recursively collect all files in this directory instead. This allows the Rule System later to reason about the files and to create the correct number of process chain branches for the sub-actions of for-each actions (see Section 3.7.3). It is also key to one of the benefits of our approach—namely that the JobManager does not require a priori design-time knowledge. The number of instances of a specific processing service can depend on the output of a previous one. Other workflow management systems require all variables to be available at design time (see Section 3.2.2).

The following is a list of endpoints and operations the Processing Connector's HTTP interface supports.

POST process chain — Endpoint: /processchains

This endpoint can be used to send a process chain to the Processing Connector. The chain will be validated and then scheduled for immediate execution. The HTTP interface will return a unique identifier with which the process chain's status and results can be queried.

Parameter	**Description**
body	The process chain that should be executed.

Status code	**Description**	**Response body**
202	The process chain was accepted and is now scheduled for immediate execution.	The process chain's ID.
400	The provided process chain is invalid or incompatible to the compute node (e.g. because of missing requirements).	None.

Status code	Description	Response body
500	Internal server error (e.g. if the process chain could not be scheduled or if another error has happened).	None.

GET process chain status and results Endpoint: /processchains/:id

With this endpoint, a client can get information about the status of a process chain. If the chain was executed successfully, the response will also include the results (i.e. the files written by the executed processing services). The status values that can possibly be returned are the same as the ones in the workflow model (see Section 3.6.1 and in particular Table 3.2).

Parameter	Description
id	The process chain's ID.

Status code	Description	Response body
200	The operation was successful.	A JSON object (see details below).
404	The requested process chain is unknown.	None.
500	The process chain's status and results could not be retrieved (e.g. because of an I/O error).	None.

The operation's response is a JSON object containing the process chain's status and—if it was executed successfully—its outputs. The outputs are represented by a JSON object whose keys are IDs of process chain arguments and values are arrays of names of written files. The following is an example response:

```
{
  "status": "SUCCESS",
  "output": [{
    "id": "argument1",
    "value": [ "/path/to/point_cloud.las" ]
  }, {
    "id": "argument2",
    "value": [
      "/tmp/output1.json",
      "/tmp/output2.json"
    ]
  }]
}
```

3.8 Fault tolerance

Due to the characteristics of distributed systems, there are many sources for potential failures (Deutsch, 1994). A good strategy to create a stable and resilient system is therefore not to try to avoid failures but to embrace them (Robbins et al., 2012). This means the system should be designed to be able to cope with failures and to continue to operate and provide service (maybe only partly) until the failure has been resolved (Nygard, 2007).

To summarise the component descriptions from the previous sections, we implemented measures to make the JobManager resilient and tolerant to the following common failures:

Chain reactions and cascading failures. A failure caused by one component can often cascade through other components in a distributed system and cause further failures. For example, if service A becomes unreachable, service B can become unavailable too because it just sent a request to A and is now waiting for an answer. This can lead to another service C becoming unresponsive because it needs to wait for B, etc.

Chain reactions and cascading failures are often caused by blocked threads and synchronous calls to remote resources. We avoid such situations, because the JobManager works *asynchronously* and *event-based*. All components continue to operate and stay responsive even if a component they depend on becomes unavailable.

In addition, the microservice architectural style allows us to implement the *bulkhead pattern* (Nygard, 2007, p. 96). The individual components (i.e. microservices) have a high coherence and are loosely coupled. They run in separate processes and often on separate virtual machines. If one of the services crashes or if one virtual machine becomes unavailable, the rest of the system is not affected and can continue to operate.

Slow responses. Every request made in a distributed system consumes resources, even if it is asynchronous. A service that responds slowly can become a source of error, in particular if the response is actually a failure and the service that performed the request unnecessarily had to keep resources.

In the JobManager we use *timeouts* to abort asynchronous operations. However, if possible we try to *fail fast* which means we try to identify and propagate a failure as quickly as possible. If there is a problem in the Processing Connector or in the Process Chain Manager, the status of affected process chains will immediately be set to `ERROR`. The Rule System does not produce process chains for erroneous outputs and the Controller aborts the workflow execution if there are no more process chains to execute. This approach has two benefits: resources are not occupied longer than necessary, and the users are informed early about problems, so they do not have to wait for results of failed workflows.

Unreachable services. In a distributed environment a service may become unreachable for different reasons: network failures, crashed service, crashed virtual machine, etc. If the problem is only temporary the service will become available again after a short period. This also applies if the failure has been detected and the service or the virtual machine was restarted (or otherwise repaired). The usual strategy to detect if a service is working again is to retry a failed call several times after short delays. However, if a service is known to be currently unavailable further calls to it are useless and should be postponed until it can be reached again.

A strategy that implements this is the *Circuit Breaker pattern* (Nygard, 2007, p. 93). A circuit breaker is a finite state machine that can have three states: closed, open and half open (see Figure 3.9). The default state is closed which means service calls are allowed. As soon as the circuit

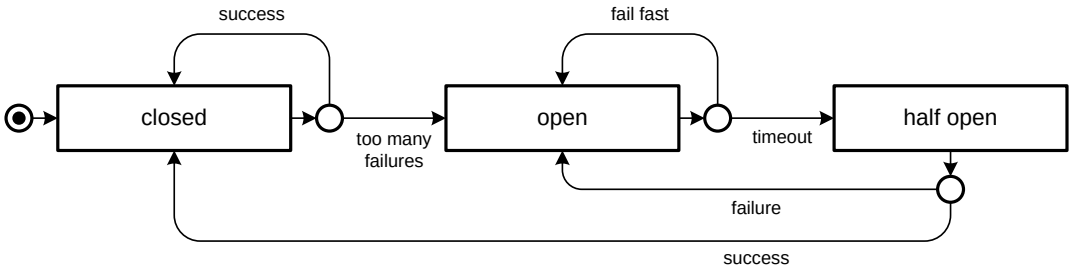


Figure 3.9 Petri net for the Circuit Breaker pattern

breaker detects a configurable number of failures, it goes into the open state. In this state service calls are blocked (they fail fast) and do not occupy further resources (the service is known to be unavailable). The circuit breaker stays in this state for a configurable amount of time and then goes into the half-open state. If the first service call succeeds in this state, the circuit breaker will return to the closed state (the service is available again). Otherwise, it will go back to the open state and block calls.

We use the Circuit Breaker pattern in various places of our architecture. The most notable one is the Process Chain Manager where we detect unreachable compute nodes with it and avoid further calls to them. As we will see in Section 5.3.3, this helps us to recover from failures and to successfully execute a workflow even if some compute nodes are temporarily unavailable.

Single Point of Failure. The JobManager can become a Single Point of Failure (SPOF) in our system. This means that if the JobManager fails the whole system will not work as expected. In order to avoid that the JobManager becomes an SPOF, we have to deploy it redundantly. As described in Section 3.4 we used Vert.x for our implementation. This tool-kit allows microservices to run in clustered high-availability mode. This means that the JobManager can be distributed to multiple processes on separate virtual machines. Vert.x takes care of connecting these processes to one virtual application.

Since we split up the JobManager into multiple independent verticles (HTTP Server, Controller, Rule System, and Process Chain Manager), we can deploy multiple instances of them to separate virtual machines. If one of the verticles fails, another instance will take over. If one of the virtual machines crashes, there will still be another one to handle incoming requests. In high-availability mode, Vert.x will take care of keeping the number of verticle instances the same all the time. If one of the verticles becomes unavailable, Vert.x will start a new instance on another virtual machine.

The JobManager's application state (workflows, process chains and their statuses) is completely stored in a database. We do not have to take special care to transfer work from a failed verticle to another one. The Controller and the Process Chain Manager regularly poll the database. If one of the verticles needs to take over from another one, it will read the state from the database automatically the next time it polls it.

System crash. The fact that we store the application state in a database also helps us to recover from system crashes. If the whole system fails—e.g. due to a power outage in the data centre—we can continue to execute workflows as soon as the system is up and running again. The Controller and Process Chain Manager regularly poll the database. After the JobManager has started, regular polling begins immediately. If the Process Chain Manager finds a process chain in the database that has not finished yet it will automatically execute it.

Work that has been done before—e.g. if half of the process chain was already executed—is not cached. The whole process chain will be executed again. However, since the processing services are idempotent (see Section 2.6) the overall result of the workflow will still be correct.

In order to make sure that this mechanism actually works, the initial request from the client to create a new workflow is only answered with HTTP status code 202 (Accepted) if the workflow was successfully stored in the database (see HTTP POST operation in Section 3.7.1).

3.9 Scalability

The JobManager has to work reliably under a high workload and regardless of how much data should be processed. Bondi (2000) lists a number of factors that can influence the behaviour of a system and thus affect its ability to scale. In our case, we have to consider the following factors:

Number of users (Load Scalability). Our system should be able to handle multiple users and work properly regardless of how many users access it at the same time. On the one hand, we achieve responsiveness through our approach to implement the JobManager in an event-based and asynchronous manner. On the other hand, our system is elastic and can handle a growing number of users (see Section 3.10).

Data size (Space Scalability). The JobManager does not process geospatial datasets itself but delegates work to processing services. It therefore does not depend directly on data volume. However, geospatial datasets are typically split into smaller files that are processed individually. Workflows can become very large, in particular if they contain many for-each actions iterating over a large number of files. To accommodate this, the Rule System splits workflows into smaller process chains. As described in Section 3.7.2 the Controller calls the Rule System in a loop until it returns no more process chains. This way, the number of chains held in memory at the same time can be kept at a reasonable size.

Number of processing services and compute nodes (Structural Scalability). The JobManager should be able to handle many processing services in different versions. It should also be able to distribute work to many compute nodes in the Cloud. In fact, these two factors could become an issue in the JobManager since both the list of processing services and the list of compute nodes are kept in memory. However, the number of items in these lists is predictable and finite. They also do not occupy very much memory. We did not experience problems with this in practise.

Data location (Speed/Distance Scalability). A distributed system should be able to work properly, regardless of where data and computational resources are located and how long it takes to transfer information. We accommodate this with our event-based asynchronous approach. The JobManager does not block when performing asynchronous requests. It always stays responsive. Responses from asynchronous operations are handled as soon as they arrive.

3.10 Elasticity

The JobManager works in an environment that is highly dynamic. Compared to a Cluster or a Grid, resources in a Cloud can be automatically provisioned and de-provisioned depending on the current workload. This property is called *elasticity* (Herbst, Kounev, & Reussner, 2013).

Elasticity can become important in the JobManager as soon as there are multiple users working with the system at the same time and/or if these users have time constraints—such as in the urban use case illustrated in Section 1.8, or in an emergency case. The JobManager should be able to handle a growing number of users and concurrent workflow executions. It should also be able to meet time constraints by making use of additional Cloud resources if necessary.

Growing number of users and concurrent workflows. The JobManager is implemented in an event-based asynchronous way. It can handle a large number of requests and always stays responsive (see Section 3.9).

As described in Sections 3.7.2 and 3.7.4 the Controller as well as the Process Chain Manager do not keep transient state but regularly poll a database for running workflows and process chains. This allows them to handle multiple workflow executions concurrently.

As described in Section 3.4, the JobManager is implemented in Vert.x. This framework enables us to distribute multiple instances of the JobManager's components to several virtual machines in the Cloud. If the workload becomes too high, we can add more virtual machines and more instances dynamically during runtime. Since the JobManager does not keep transient state, we can even do this during the execution of workflows.

Time constraints. If there are constraints that require the JobManager to finish certain workflows in a given amount of time, regardless of how many other workflows are currently running, we can increase the number of compute nodes dynamically. The JobManager uses the Vert.x Config module with which an application's configuration can be changed during runtime. This allows us to add and remove compute nodes, even during the execution of workflows.

Initially, we planned to scale out automatically and to let the JobManager request more Cloud resources if necessary based on configurable rules. However, the rules would have to be implemented very carefully. Otherwise, this approach could have led to excessive use of resources and to high costs. We therefore propose to handle this issue on the level of the Cloud infrastructure. Most Cloud administration consoles have a way to allocate more resources based on given metrics and based on an available budget. Through the configuration mechanism described above, new computational resources can be made available dynamically to the JobManager.

3.11 Supported workflow patterns

In order to summarise the functionality of our system in terms of workflow management, we revise the list of workflow patterns defined by van der Aalst et al. (2003) and Russell et al. (2004b, 2004a, 2006, 2016) (see also Section 3.2.2). In the following table we list each pattern our system supports and how we implement it. This summary can be used as a reference to relate our approach to other workflow management systems.

Pattern	Description	Implementation
Control-flow patterns		
WCP-1	*Sequence* – Execute tasks sequentially	Our workflow model allows actions to be chained through output/input parameters. Actions will only be executed if all inputs are available and this can only be true if the preceding actions have finished and written their output.
WCP-2	*Parallel Split* – Execute multiple workflow branches in parallel	Section 3.7.3 shows how process chains can be split and executed in parallel.
WCP-3	*Synchronisation* – Join multiple parallel workflow branches	Similar to WCP-2: see Section 3.7.3. Process chains with predecessors will only be executed if all preceding process chains have finished successfully.
WCP-11	*Implicit Termination* – Terminate workflow when there is no more work to be done	The Controller will finish the workflow execution if the Rule System does not produce any more process chains (See Section 3.7.2).
WCP-12	*Multiple Instances Without Synchronisation* – Multiple instances of the same task can be created. Synchronisation is not necessary.	Our for-each action creates multiple instances of its sub-actions which can be executed independently (see Section 3.6.1).
WCP-13	*Multiple Instances With a Priori Design Time Knowledge* – The number of instances of all actions is already known before the workflow is executed.	Given one of the variables in our workflow model has a `ListValue` and the for-each action is applied to this variable, then the number of all instances of the for-each action's sub-actions is known in advance.

Pattern	Description	Implementation
WCP-14	*Multiple Instances With a Priori Runtime Knowledge* – The number of instances of all actions can be decided at runtime before these actions are executed.	Given an action writes a number of files into an output directory, the Processing Connector lists the directory contents and passes the files as available inputs to the Rule System. Subsequent foreach actions can then be applied to this file list.
WCP-20	*Cancel Case* – The workflow execution can be cancelled.	The JobManager's HTTP interface allows workflows to be deleted from the database. In such a case the Controller will produce no more process chains for the deleted workflow. The Process Chain Manager will finish the execution of currently running process chains, but stop (or return to standby) as soon as there are no more process chains to be executed in the database.
Resource patterns		
R-RF, R-CE	*Retain Familiar* – Allocate a workflow action to the same resource as a preceding one, *Chained Execution* – Immediately execute the next action when the preceding one has finished.	The Rule System creates process chains containing one or more directly connected processing services. It splits them into smaller chains using the constraints described in Section 3.7.3. In order to leverage data locality, all services of a process chain are executed on the same compute node.
R-CBA, R-SHQ	*Capability-based Allocation* – Allocate actions based on capabilities offered by a resource, *Shortest Queue* – Allocate an action to the resource with the shortest work queue.	The Process Chain Manager uses a *Capability-based Shortest Queue* algorithm (see Section 3.7.4).
R-DBAS	*Distribution by Allocation – Single Resource* – Directly allocate an action to a resource.	The Process Chain Manager sends a process chain to a single compute node for execution. It only tries another one if the node is currently unavailable. Once the process chain has successfully been transferred to the node, it is expected to be executed. No other node will be tried, even if the execution fails.
R-DE, R-LD	*Distribution on Enablement* – Actions are allocated to resources as soon as they are ready for execution, *Late Distribution* – Allocate resources at a later point in time.	As soon as the Controller has stored a process chain in the JobManager's database, it can be picked up by the Process Chain Manager. If there is a compute node available the Process Chain Manager immediately allocates the process chain to it. Otherwise, it retries the operation later.
R-D	*Delegation* – A resource delegates work to another one.	In our case this can happen for environments such as Spark or Hadoop clusters. The compute nodes registered with the JobManager are those that run the cluster manager, but not necessarily those that do the work. The cluster manager may distribute work to other nodes in the Cloud unknown to the JobManager.

Pattern	Description	Implementation
R-E, R-SD	*Escalation* – An action is allocated to another resource than the one it has been allocated before, *Deallocation* – An action is deallocated from a resource.	The Process Chain Manager can allocate a process chain to another compute node if the selected one is currently unavailable. Special care is taken to prevent the process chain to be allocated to the same compute node again.
R-CA	*Commencement of Allocation* – Resources execute allocated actions immediately.	The Processing Connector starts the execution of a process chain immediately after is has received it.
R-SE	*Simultaneous Execution* – A resource can execute multiple actions at the same time.	The Processing Connector can run multiple process chains in parallel to make best use of available resources.
Data patterns		
D-1	*Task Data* – Data elements can be defined by task	Workflow actions can have parameters.
D-2	*Block Data* – Data elements can be defined by block (or sub-workflow)	Our for-each actions have sub-actions. All of them can access the data the for-each action is applied to. A list of sub-actions basically is a sub-workflow. In addition, in Chapter 4, *Workflow Modelling* we introduce `with` blocks which directly implement this pattern.
D-3	*Scope Data* – Some data elements are only accessible by a subset of tasks.	There is no direct notion of a scope in the JobManager. The Domain-Specific Language we introduce in Chapter 4, *Workflow Modelling*, however, has variable scopes.
D-4	*Multiple Instance Data* – Every instance of a task can maintain its own data.	We support this when we execute a processing service in a separate environment such as a Docker container. In this environment a service can create arbitrary files without interfering with other services or service instances.
D-5, D-6	*Case Data* – All tasks in a workflow instance can access the same data, *Workflow Data* – Data can be shared between workflow instances.	Workflow inputs and outputs are stored in a distributed file system and persist across workflow executions. Tasks in a workflow can access the same input data. Output data of one workflow can become input data of a subsequent one.
D-7	*Environment Data* – Tasks can access files in the external operating system.	This is supported, for example, if we execute a processing service in a Docker container. Through the service metadata attribute 'runtime_args' (see Section 3.6.2) we can mount files or directories from the external operating system to the container. Also, if we execute processing services directly without a container, they can access any data in the local file system.
D-8	*Data Interaction – Task to Task* – Data elements can be passed from one task to another.	Actions in our workflow model are connected through output and input variables. The same applies to executables in the process chain model.

Pattern	Description	Implementation
D-9, D-10, D-11, D-12	*Data Interaction – Block Task to Sub-Workflow Decomposition*, *Sub-Workflow Decomposition to Block Task*, *to Multiple Instance Task*, *from Multiple Instance Task*	These patterns are implemented by our for-each action. We can pass data to this action which then distributes it to its sub-actions. The results of the sub-actions is collected and can then be passed to subsequent actions.
D-14, D-15	*Data Interaction – Task to Environment – Push-Oriented*, *Environment to Task – Pull-Oriented* – A task can pass data to resources or services in the environment, and can request data from them.	Processing services write to the distributed file system and read from it.
D-24	*Data Interaction – Environment to Workflow – Push-Oriented* – Environment data can be passed to a workflow.	This relates to the datasets a workflow should be applied to. These are basically the input variables in the workflow model which are defined at the time the workflow execution starts.
D-26	*Data Transfer by Value – Incoming* – Workflow components may receive input by value.	Actions in our workflow model can have parameters which are defined by variables and values.
D-28	*Data Transfer – Copy In/Copy Out* – A workflow component copies data into its address space and writes back the final results.	This is the usual case with our processing services. They read datasets from the distributed file system, transform it, and write back the results.
D-29, D-30	*Data Transfer by Reference – Unlocked and With Lock* – Data can be communicated between workflow components by reference without copying the data.	We use file names to communicate the location of datasets to the workflow actions. Whether concurrent data access is protected by a lock or not is undefined, but variables in the Domain-Specific Language in Chapter 4, *Workflow Modelling* are immutable. Therefore locks are not required, because there only are concurrent reads. Writes always happen exclusively to new files.
D-31, D-32	*Data Transformation – Input and Output* – The possibility to transform input and output data prior or after the execution of a workflow component.	As described in Section 3.7.3 the Rule System may insert additional processing services before or after a service in a process chain to transform its input or output. This depends on the service metadata. For example, the Rule System may add a data conversion service between service A and B if A's output data type is incompatible to B's input data type.

3.12 Summary

In this chapter we discussed how our system performs distributed data processing based on workflows. We first described background on workflow management systems and then compared our approach to related work. We then gave an overview of the software architecture of our workflow management service, the JobManager, and then described its internal data models and

components in detail. Finally, we discussed fault tolerance, scalability as well as elasticity, and summarised the functionality of our system using well-defined workflow patterns.

One of the main benefits of our JobManager is that it can execute processing services with arbitrary interfaces. We presented a lightweight approach to describe the interface of a service with JSON-based service metadata. Based on this metadata, the JobManager can generate command-line calls for the services and orchestrate them to workflows. Developers can integrate their services into our system without fundamental modifications. They just need to create an interface description. The JobManager takes care of deploying the services and parallelising them in the Cloud.

Our approach to workflow management has a significant advantage over other systems that support the execution of workflows in a distributed environment. Since the number of instances of a processing service can, in our case, depend on the results of a preceding one, we support dynamic workflow execution without a priori design-time knowledge. Other workflow management systems require complex workarounds to implement this behaviour.

Internally, the JobManager makes use of rule-based system that is configurable and allows us to adapt the workflow execution to different domains. A typical scenario from the geospatial domain is, for example, that a certain user does not have access to a data set or a processing algorithm because of a missing licence. We can accommodate for this by adding a rule to our rule-based system to prevent the data set from being used as input or to disallow using the processing algorithm (and probably find an alternative). Another example from the geospatial domain is that users often prefer specific data sets or algorithms over others (e.g. due to better quality, better reputation, etc.). Such preferences are user-specific and can be implemented with our rule-based system.

In the following chapter we further discuss the configurability and adaptability of our system to specific use cases. We present a Domain-Specific Language with which users can control the behaviour of our system by specifying processing workflows.

4

Workflow Modelling

In the previous chapters we focused on the Cloud architecture and how we can control the processing of large geospatial data in a distributed system. An additional challenge lies in providing the user or domain expert with means to control the data processing in an easy and understandable manner.

In this chapter we present a way to define a processing workflow in a *Domain-Specific Language (DSL)*. A DSL is a lightweight programming language tailored to domain users with little to almost no IT background. Such a language uses the vocabulary of a specific domain and is hence easy to understand and learn for people working in this domain.

The chapter is structured as follows. First, we motivate the use of Domain-Specific Languages for distributed geospatial data processing and give an overview of related work. We then present a novel method for DSL modelling, followed by two example use cases demonstrating how it can be applied in practise. After that, we describe a technique to map DSL expressions to processing services and datasets in the Cloud. The chapter ends with a summary and conclusion.

4.1 Introduction

Current desktop-based GIS solutions are feature-rich and suitable for a wide range of geospatial applications. The commercial software ArcGIS by Esri and the Open-Source tool QGIS, for example, are among the most popular desktop GIS solutions and used by many companies, municipalities and other institutions. Whenever users need more flexibility than these products already offer, they can develop scripts and extensions. This also allows them to automate recurring workflows which have otherwise to be performed manually. The scripts and extensions are usually written in a general-purpose programming language such as Visual Basic, C++ or Python. These languages provide high flexibility, but they also require the users to have experience in programming. GIS users are experts in their domain but typically have no background in computer science. Writing scripts is often a complex and tedious task for them.

This problem becomes even more complex in a Cloud Computing environment where users are faced with additional questions such as the following:

- Which algorithms do I have to use? Which algorithms can actually be executed in my Cloud environment?
- How do I model my problem so it can be solved with a parallelised algorithm?
- Does my data have to be partitioned in order to be processed in parallel in the Cloud? How small or large do I have to make the chunks in order to process them efficiently and to exploit the processing capabilities of the Cloud as much as possible (granularity)?
- How do I have to store my data so the parallelised algorithm can access it efficiently? Where do I have to store it (i.e. in which virtual data centre)?
- etc.

Current Cloud technologies give help on these questions only to a certain degree. Users are mostly required to find solutions on their own. Approaches such as MapReduce (Dean & Ghemawat, 2008) allow for processing large amounts of data, but defining a MapReduce job—let alone a chain of several jobs that make up a complete workflow—requires a deep understanding of the processed data, the algorithms, and the infrastructure (i.e. the Cloud environment). Additionally, users have to have a background in computer science and software development, in particular functional programming.

Users dealing with geospatial data processing need to be able to define high-level workflows that focus on the actual problem and not the technical details of the underlying, executing system. The workflow definition language should not be a general-purpose programming language requiring comprehensive programming skills. Instead, it should be designed for a single application domain, which means it should consist of a reduced vocabulary that is tailored to the tasks the domain expert needs to perform. In particular, such a language should have the following characteristics:

- It should be tailored to describe workflows (and nothing else).
- It should be tailored to a certain application domain—in this case geospatial processing, or even more specific, data processing in urban planning (see Section 4.4), land monitoring (see Section 4.5).

A language that is meant to be used in a specific application domain is called *Domain-Specific Language* or *DSL*. According to Fowler (2010) there are two kinds of DSLs: internal and external ones. *Internal DSLs* are based on another programming language (the host language) which is typically a general-purpose language. In its simplest form an internal DSL is nothing more than a well-designed API like a fluent interface that makes it easy to express a complex matter in a readable way. More advanced internal DSLs make use of special features of the host language such as AST transformations—in the case of Groovy DSLs for example (Dearle, 2010)—or operator overloading and implicit values—see Scala DSLs (Hofer & Ostermann, 2010). These features allow language developers to create an internal DSL that looks like a completely new programming language, but reuses components of the host language such as its lexer and parser (Hudak, 1998). Internal DSLs are very powerful because they are simple and easy to use, but also allow their users to fall back to using the host language if they need more advanced features. A good example for this is the Domain-Specific Language interface of the build automation tool Gradle (Gradle Inc., 2017). Almost all tasks necessary to specify a complex build process can be expressed with Gradle's internal DSL. Whenever users need more flexibility, they can write code in the host language Groovy.

An *external DSL*, on the other hand, can be designed independently of existing languages. It has a separate syntax and grammar and requires a new interpreter or compiler. External DSLs do not offer the possibility to fall back to features of a host language and are often not as powerful as

internal ones. Nevertheless, their limitations enable the interpreter or compiler to produce optimised code. For example, a language that does not offer a way to change the value of a variable or to do I/O operations may be better suited for writing a multi-threaded program than an imperative general-purpose language where developers need to take special care of possible side effects.

A comprehensive survey on Domain-Specific Languages is given by Mernik, Heering, & Sloane (2005). They describe patterns in the decision, analysis, design, and implementation phases of DSL development and try to give support to developers on when and how to develop such languages. One of the issues they discuss is the question of whether a language designer who wants to solve a particular problem should create an internal DSL (or *embedded DSL*) or if an external DSL is a more suitable approach. In this regard, they state the following:

> If the DSL is designed from scratch with no commonality with existing languages (invention pattern), the recommended approach is to implement it by embedding, unless domain-specific analysis, verification, optimization, parallelization, or transformation (AVOPT) is required, a domain-specific notation must be strictly obeyed, or the user community is expected to be large.

In this work we aim for creating a Domain-Specific Language that is easy to learn for non-IT personnel and that requires no background in programming. We therefore want to design a new language that does not resemble existing general-purpose programming languages. This is what Mernik et al. call the *invention pattern*. According to them, embedding is the recommended approach for this. Consequently, we should design an internal DSL. However, Mernik et al. state that an external DSL is more suitable in at least one of the following cases:

- If *domain-specific analysis, verification, optimisation, parallelisation, or transformation (AVOPT)* is required. In this work we want to create a system that processes geospatial data in a distributed manner in the Cloud. Our JobManager applies domain-specific analysis as well as optimisation and parallelisation to schedule the execution of processing services. As discussed above, this would be barely possible if we allowed access to all features of a general-purpose language. The limitations of an external DSL actually enable the JobManager to calculate a suitable execution plan.

- If a *domain-specific notation* must be strictly obeyed. This is not required by our use cases but it is beneficial as it makes the language more readable and easier to learn for domain experts.

- If the *user community* is expected to be large. In this work we target GIS experts from various organisations, municipalities and institutions. We cannot assume that the users will have a background in programming. Mernik et al. specifically state that error reporting is a problem with internal DSLs as the error messages depend on the host language compiler and cannot be customised or are in the worst case misleading. A separate interpreter or compiler for an external language, on the other hand, can be customised to produce domain-specific error messages that can be understood by a wide range of users.

In this sense, we focus on external DSLs. This allows us to design a language independently of any implications of a host language. It also enables us to create a DSL that is tailored to describing workflows, but does not offer any features that would prevent execution in a distributed environment. In the following, the term *DSL* is hence used synonymously with *external DSL*.

Mernik et al. (2005) refer to a large number of earlier articles and argue that while many of them focus on the development of particular DSLs there is almost no work on DSL development methods. In Section 4.3 we follow up on this matter and describe a method for DSL modelling. We apply it in Section 4.4 to an example use case to create a language tailored to the definition of geospatial processing workflows.

4.2 Related work

In this section we discuss related work on Domain-Specific Languages, in particular their use in software development and Cloud Computing. We also compare textual languages with visual programming languages and discuss methods to model and develop DSLs.

4.2.1 Domain-Specific Languages in software development

Domain-Specific Languages have a history in software development. They are used in various areas of software systems, in particular in those that depend on user requirements, business workflows, or on the environment where the system is deployed. Figure 4.1 depicts the layers of code in a complex software architecture, each of them being defined by their dynamics and programming languages used (see Bini, 2008).

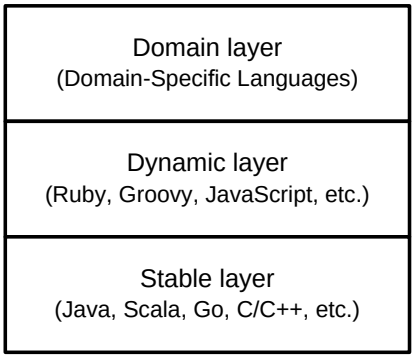


Figure 4.1 Layers of code in a complex software architecture

The stable layer contains code that rarely changes. It is usually written in a statically typed language such as Java or C++ and contains the application's kernel. The second layer changes more often. Here, developers use scripts written in dynamically typed languages such as Ruby or Groovy to affect the application's behaviour and to customize it for a certain group of users or customers. In the third layer, the domain layer, users can change the application's behaviour directly through scripts written in Domain-Specific Languages. This layer is the one that changes most often. It is directly connected with the users' requirements and the tasks they need to perform in their respective application domain.

The advantage of using dynamic languages in the second layer is that developers do not have to recompile the whole application if their workflow has changed or if the application should be used for another customer working in the same area but having slightly different requirements. The code in both the dynamic layer and the static layer is typically written by software developers and not end-users. The third layer, on the other hand, is about the domain the application is used in. Even in the same domain, users often need to perform various tasks with different workflows and requirements. For example, in the geospatial domain, as described earlier, users need to process heterogeneous geodata with a range of algorithms and processes. It would be too expensive and time-consuming to ask the application developers to extend the system every time a new task has to be done. Instead, in the domain layer, the users define workflows on their own. Obviously, they

need an easy and understandable interface for that. This is why in the third layer Domain-Specific Languages are used, which are tailored to the tasks the users have to perform in their application domain.

4.2.2 Domain-Specific Languages for data processing

The use of DSLs in the area of Cloud Computing to control distributed processing has been demonstrated by Olston et al. who present the software library Apache Pig and in particular its high-level processing language Pig Latin (Olston, Reed, Srivastava, Kumar, & Tomkins, 2008). The language looks a lot like the database query language SQL—which is in fact also a Domain-Specific Language—so users who are familiar with SQL are able to quickly learn and use Pig Latin as well. For example, the following script loads a 3D point cloud including information about classification represented by a number specifying if a point has been classified as belonging to the ground, a building, a tree, or if the classification was inconclusive (in this case the number will equal `-1`). The script removes points that are not classified and then groups the remaining points by their classification.

```
point_cloud = LOAD 'point_cloud.xyz' USING PigStorage(',')
    AS (x: double, y: double, z: double, classification: int);
filtered_points = FILTER point_cloud BY (classification != -1);
grouped_points = GROUP filtered_points BY classification;
DUMP grouped_points;
```

The language drives distributed MapReduce jobs executed in the Cloud by the Apache Hadoop framework. Pig Latin simplifies the process of specifying complex parallel computing tasks which can sometimes be tedious even for experienced programmers. However, it is very generic and does not target a specific application domain like the language we aim for in this work. In addition, it is not really a workflow description language but can be better compared to a query language. It requires exact knowledge about the structure of the data to process.

Pig as well as Pig Latin lack support for geospatial processing. This gap is closed by the SpatialHadoop framework which adds spatial indexes, as well as geometrical data types and operations to Apache Hadoop (Eldawy & Mokbel, 2013). In addition, SpatialHadoop offers a DSL based on Pig Latin. Pigeon is a high-level query language that allows users to specify queries that operate on geospatial data (Eldawy & Mokbel, 2014). However, since Pigeon is based on Pig Latin it shares the same properties and is therefore quite different to the language that we design in this work.

Domain-Specific Languages have also been applied successfully in the area of parallel computing. Chafi et al. (2011) present Delite, a compiler framework and runtime for high performance computations. They specifically target multi-core systems as well as GPGPUs (general-purpose computing on graphics processing units). Their approach has been successfully applied to OptiML which is a Domain-Specific Language for high-performance parallel machine learning (Sujeeth et al., 2011). DSLs developed with Delite are embedded into the Scala programming language. The advantage of this approach is that existing compiler components can be reused (Delite acts as a compiler plug-in) which substantially reduces the amount of work for the language designer. However, language users are required to have at least some experience in programming, in particular Scala whose syntax—which sometimes is referred to as being complicated and hard to read (Juric, 2010; Ulrich, 2011)—cannot be completely hidden in the DSL. In this work we develop an external DSL instead that is not coupled to a general-purpose programming language and whose syntax can therefore be designed freely.

Another area where DSLs have been used for data analysis and processing is urban planning. In a previous work we presented a Domain-Specific Language for urban policy modelling (Krämer, Ludlow, & Khan, 2013). The language offers the possibility to specify policies and policy rules in

a formalised and machine-readable way. At the same time the policy rules stay clearly readable and understandable for urban planners and decision makers. The following example shows a policy rule and a production rule that changes a visualisation on the user's screen when the analysed data indicates the policy rule was not met.

```
The number of cars on street B per day
    has to be lower than 2500.

When the number of cars on street B per day
    is higher than 2500
then display street B in red.
```

We further elaborated the idea of using production rules to process geospatial datasets in another one of our previous work (Krämer & Stein, 2014). We presented a graphical DSL that can be used to specify typical spatial processing workflows in the urban planning domain. The graphical user interface has been integrated into a 3D GIS (Geographic Information System). User evaluation carried out in the urbanAPI research project funded by the European Commission confirmed that such a DSL is indeed useful, but more work is needed, in particular since the language is not designed to operate in a Cloud Computing environment (Krämer, 2014).

4.2.3 Textual and visual programming languages

The workflow management systems currently available offer different ways to define workflows. While systems such as Pegasus rely on a textual programming language, others such as Kepler or Taverna provide a graphical user interface. In addition, there are a couple of workflow languages available (see Section 4.2.4) such as YAWL (Yet Another Workflow Language), CWL (Common Workflow Language) or BPEL (Business Process Execution Language) for which software vendors have developed textual or graphical editors.

For the user interface in this work we have considered both textual and graphical ways to define workflows. We based our decision to design a textual DSL instead of a graphical editor on the comparison of general textual programming languages (TPL) and visual programming languages (VPL). Whitley (1997) presents a survey on the state of the art in VPLs and tries to give empirical evidence for and against them. He reviews a large number of publications and compares their results in order to decide whether TPLs or VPLs are superior to each other. He concludes that visual programming languages can improve usability and help users perform specific tasks, but this does not apply in all cases. The same is true for textual programming languages. The reason for this can be attributed to the *match-mismatch hypothesis* described by Gilmore & Green (1984). They state that every notation highlights some kinds of information while it hides others. This means that while a notation might be suitable for a certain task, it will fail for others. The match-mismatch problem implies that for every task different kinds of notations have to be used (Ruppert et al., 2015). This is supported by Brooks (1987) who claims there is "no silver bullet"—i.e. no "development […] that by itself promises even one order-of-magnitude improvement in productivity, in reliability, in simplicity." He also discusses earlier research in visual programming languages and states: "Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will."

In addition to the match-mismatch hypothesis, Whitley (1997) also discusses the role of primary and secondary notation. He states that the primary notation—i.e. the syntax and grammar of a TPL or the visual elements of a VPL—is of major importance for the comprehensibility of a language, but the secondary notation—the way elements are presented on the screen—also plays a major role. The secondary notation of textual programming languages is the way the text is formatted—i.e. how the lines are indented or the text is presented (bold, italic, underlined, etc.). In

visual programming languages the secondary notation is the way visual elements (boxes, arrows, etc.) are placed on the screen. In both cases, the secondary notation influences usability. A text that is well formatted or a visual representation whose elements are well placed can help users understand a program, but badly formatted text and misplaced visual elements can even confuse them.

In our experience it is much harder for untrained users to create a clear and tidy visual representation than it is to write a well-formatted textual program (especially if the matter to express is complex). This is supported by Booth & Stumpf (2013) who present an exploratory empirical study on the end-user experience with visual and textual programming environments for the physical computing platform Arduino. They investigate how adult end-user programmers—non-professional programmers with no experience in the Arduino platform but basic knowledge of programming—can perform a number of tasks in a textual and a visual environment. They conclude that the study participants managed to finish the tasks better in the visual programming environment. However, they also note that the tasks where they performed better were mostly related to modifying an existing program instead of creating a new one. The visual layout was already there and the learning barriers were therefore much lower.

Another study on the comprehensibility of visual and textual programs is presented by Green, Petre, & Bellamy (1991). They explore the fallacy many people make by assuming a VPL is better just because it is visual. They call this phenomenon "Superlativism" because people often think that visual perception is more efficient than reading text in general, while it can be shown that some tasks benefit from visual programming languages and others do not. They conclude that (at least in their study) VPLs performed worse compared to TPLs. Gilmore and Green's match-mismatch hypothesis proved correct again because they were not able to find a single VPL with which they could cover all possibilities in their experiment. Their study is further supported by Petre (1995) who explains "Why looking isn't always seeing" and questions correctness of the English idiom "A picture is worth a thousand words". He presents figures of identical programs in visual and textual form and explains in which aspects they succeed and in which they fail. He also discusses the importance of a language's secondary notation.

Neag, Tyler, & Kurtz (2001) investigate visual and textual programming in automatic testing. They list a number of aspects that limit the use of VPLs:

- *Visualization restriction.* A computer screen has a certain size and the number of elements it can display is limited. Complex visual programs can become very hard to read if users have to scroll.

- *Limited modularization capabilities.* Textual programming languages typically provide means to structure a program into namespaces, modules, packages, objects, etc. Many visual programming languages lack these capabilities.

- *Limited support for embedded documentation.* TPLs allow comments to be added in the source code to document the program's behaviour.

- *Limited ability to model complex data structures and to describe complex algorithms.* It is hard to express both data flow and control flow in a single visual program.

In addition, we note that unless a standardised modelling language such as BPMN is used, a visual programming language binds the user to a specific software solution (vendor lock-in). Textual programs can easily be transferred from one system to another and can be edited in any text editor while visual programs require a specific modelling tool.

In summary, we can conclude that there is no clear evidence whether a TPL or a VPL is superior or not. There are many studies proving that a domain-specific representation helps users in performing certain tasks. Visual programs often perform better when the tasks are well defined and their extent is limited, but fail otherwise. Visual programs suffer from the problems described by the match-mismatch hypothesis as much as textual ones. They benefit from a good secondary

notation, although a badly structured visual program can be much harder to comprehend than a badly formatted text. Textual programming languages also tend to be more suitable for a wider range of tasks and easier to use in terms of tooling support.

In this work we focus on a textual programming language. Our language covers a specific domain (geospatial data processing) and supports many different tasks within this domain. Modularity and extensibility is therefore of major importance for us. In this chapter we present a modelling method for a textual DSL with which we can achieve these properties. Doing something similar with a visual programming language is a task beyond the scope of this work.

4.2.4 Workflow description languages

When reviewing existing workflow description languages we have to differentiate between those that are meant to describe business workflows and those that support scientific workflows (see Chapter 3, *Processing*). The Business Process Model and Notation (BPMN), for example, is a graphical language supporting business workflows (OMG, 2013). It has elements for describing activities (tasks and subtasks), events, sequences, messages, and others. Workflows designed in BPMN typically describe a control flow in an organisation where a subsequent activity may only start when another one has finished or a certain event has happened. In this chapter we aim for designing a language that supports modelling data flow. In our language, activities consume data from a source (e.g. a Cloud data store or a previous activity) and produce new data. The control flow in our language is driven by the way the data flows from one activity to another. Apart from that, BPMN is a graphical language while we aim for creating a textual one.

WS-BPEL (Web Services Business Process Execution Language), on the other hand, is a textual language supporting the modelling of business processes that are executed by web services (OASIS, 2007). It is standardised and based on XML. The current specification was defined by the Organization for the Advancement of Structured Information Standards (OASIS) and is named WS-BPEL 2.0. In contrast to BPMN, WS-BPEL also supports modelling of fully automated workflows that require no human interaction and can therefore be used to create scientific workflows. However, the language is tightly coupled with the WSDL service model (W3C, 2001). Support for processing services that do not implement a WSDL interface and that are not web services is rather complex. In addition, we aim for a language that is easy to understand for domain experts. WS-BPEL is based on XML and therefore harder to read than a language that is tailored to specific domain users.

Another language for business modelling is YAWL - Yet Another Workflow Language (van der Aalst & ter Hofstede, 2005). Compared to BPMN and WS-BPEL, it has stronger semantics and tries to avoid discrepancies between the workflow model and its actual implementation in the organisation. Since it is built on petri nets, it also enables formal business process validation. In addition, YAWL supports dynamic workflows that may change during the execution. Compared to the language we aim for in this work, YAWL models control flow instead of data flow. It is also based on XML and hard to read for domain users.

Apart from modelling languages for business workflows, there are a number of languages specifically supporting scientific workflows and designed for the modelling of distributed processes running in Grids or Clouds. The Common Workflow Language (CWL), for example, allows for writing workflows that are portable across different devices and target platforms (Amstutz et al., 2005). It supports data-driven workflows specifically from scientific domains including Bioinformatics, Physics, and Chemistry. CWL workflows are written in JSON or YAML. While these file formats are easier to read for humans than XML, the underlying data model is still rather complex. Users of CWL must have a knowledge about what processes they need to execute and what kind of inputs and outputs they have. Although it claims to be portable, the workflows one

can describe with CWL are tightly coupled to the executed processes. The portability in CWL is achieved by running processes in isolated Docker containers. Due to these aspects, the language can better be compared to our workflow model described in Section 3.6.1 than to the DSL we design in this chapter.

Another technology that has gained traction recently is the Amazon States Language (Amazon, 2016). This language is based on JSON and supports defining state machines for AWS Step Functions, a web service that facilitates orchestration of microservices in the AWS Cloud. A state machine basically models control flow and not data flow since state transitions are triggered by events and conditions—e.g. the availability of certain data, or the end of a task. The Amazon States Language requires the user to know how state machines work and how their problem can be modelled with such a machine. In addition, JSON is not as easy to read as the language we aim for in this work.

The authors of the Workflow Definition Language (WDL), on the other hand, claim that their language was specifically designed "to be read and written by humans […] Without requiring an advanced degree in engineering or informatics." (Broad Institute, 2017). However, WDL resembles JSON and has a rather complex data model. It also requires knowledge about the processing services to be executed. If a new service should be added to the language the user must define a 'task' which is a description of the service's input and output parameters, but may also contain information about how the service should be executed—e.g. in a Docker container. In WDL task definitions are often intermixed with the workflow description which leads to a high coupling between workflow and the processing services. In contrast, the approach we present in this work aims for decoupling the workflow description from both the executing infrastructure and the services. In Section 4.6 we describe a rule-based way to map terms in our language to processing services. This mapping is not intermixed with the workflow script but a configuration the system developer or administrator creates. In addition, as shown in Chapter 3, *Processing* our services are described by metadata files in JSON format which are created by the service developers and not the users writing the workflow.

4.2.5 Domain modelling methods

Domain modelling is often used in software engineering to identify major concepts in a domain and to find a suitable software design (see Evans, 2003). A domain model consists of conceptual classes which are not necessarily software classes. Instead, domain models are typically used by software architects to communicate the conceptual structure of a system design to the domain experts as well as the developers, and to better understand the use cases and requirements in the analysed domain. The modelling method for Domain-Specific Languages we present in Section 4.3 also makes use of domain models. They help the language designer to better understand the concepts used by domain experts and to create a language based on a vocabulary they are familiar with.

Software engineering methods can also be applied to other modelling tasks. De Nicola, Missikoff, & Navigli (2009), for example, present an approach named UPON (Unified Process for ONtology building). They use this approach to build ontologies in the domains of automotive, aerospace, kanban logistics, and furniture. UPON is based on the *Unified Software Development Process*—also known as the *Unified Process* or *UP* (Jacobson, Booch, & Rumbaugh, 1999)—and is therefore use-case driven, iterative and incremental. Analysing use cases helps De Nicola et al. build ontologies that target a certain application area. In UPON, ontology developers work closely together with domain experts and frequently review their results in an iterative way. The ontology is then incrementally extended. De Nicola et al. conclude that their method is highly scalable and customisable. Their research shows that the use of software engineering methods outperforms existing approaches to ontology modelling.

UPON is similar to our DSL modelling approach. We also propose an iterative process that is based on best practises from software engineering. Our DSL modelling method also involves domain experts which helps us create a language that fits their needs.

In a later article, De Nicola & Missikoff (2016) present a lightweight version of UPON called UPON Lite. The main benefits of their new approach is that it is simpler and puts domain users into focus so that they can mostly perform ontology modelling on their own without the help of ontology experts. UPON Lite encompasses six steps in which the domain users first collect a number of terms that characterise their domain before they are put into relation.

UPON Lite and our DSL modelling approach are similar to a certain degree. The differences are as follows:

- In our DSL modelling approach we also identify relevant domain terminology, but instead of asking the users to create a list of terms from scratch, we first collect user stories and then apply text analysis.
- We do not focus very much on the meaning of terms (and their relations) because we are more interested in the terms themselves instead of their semantics.
- The outcome of our approach is not an ontology but a grammar for a Domain-Specific Language.

Applying methods from software engineering to both ontology modelling and DSL modelling is novel. De Nicola et al. show that these methods are suitable for building ontologies. In the following, we do so for Domain-Specific Languages.

4.3 DSL modelling method

In order to design the Domain-Specific Language for distributed geospatial data processing, we propose a novel incremental and iterative modelling method consisting of the following steps:

1. Analyse the application domain
2. Create user stories
3. Analyse user stories and look for relevant subjects, objects, adjectives, and verbs
4. Create a domain model using subjects and objects found in the user stories as classes
5. Identify relevant verbs which become actions in the Domain-Specific Language
6. Build sample DSL scripts based on the modelled domain
7. Derive formalized grammar from the sample DSL scripts
8. Review and reiterate (go back to step 1) if needed

The first couple of steps are inspired by object-oriented software engineering. Steps 1 and 2 are essential for every modern agile and lean software development project. They belong to the general requirements analysis phase where software developers and stakeholders (most likely domain users) work together to identify functional and non-functional requirements. This typically results in a

number of user stories describing how the domain users would like to interact with the system. Ideally, these stories are created by the users themselves. This ensures they are written in "the user's own words", which basically means they use the exact same vocabulary the domain users are used to from their everyday work. Based on this, a Domain-Specific Language can be created that contains terms from the application domain and that is hence easy to understand and learn.

The text analysis performed in step 3 of the DSL modelling process provides the basis for the domain vocabulary. The collected objects, subjects and verbs have to be structured and classified, so a machine-readable programming language can be defined. In step 4, a domain model is created describing the relations between subjects and objects. Additionally, in step 5 the relevant verbs are identified that will later become actions in the Domain-Specific Language.

In step 6 sample scripts written in the (not yet formalised) DSL are created. Just like in the first two steps, the domain users should be involved here to provide feedback on the sample scripts. This ensures the final language will look as expected. After that, a formalised grammar is created in step 7. This makes the language machine-readable and interpretable.

The whole modelling process is iterative. The result is reviewed and revised if necessary in close collaboration with the domain users. In the following we demonstrate our DSL modelling method by applying it to two example use cases. The use cases are described in detail in Section 1.8. We only repeat relevant parts here.

4.4 Use case A: Urban planning

This section focuses on the use case described in Section 1.8.1 dealing with urban planning. We demonstrate how a Domain-Specific Language tailored to experts from this domain can be created. The use case includes tasks from the typical work of an urban planner who needs to integrate and process geospatial data from different sources to create products such as topographic maps, orthophotos, and 3D city models.

The *first step* in our DSL modelling method is the domain analysis. As a result of the domain analysis, we create a number of user stories (*step 2*). We refer to Section 1.8.1 where we have already described and analysed the use case in detail. For reasons of clarity and comprehensibility, we repeat the user stories here:

> **User story A.1:** As an urban planner, I want to capture topographic objects (such as cable networks, street edges, urban furniture, traffic lights, etc.) from data acquired by mobile mapping systems (LiDAR point clouds and images) so I can create or update topographic city maps.

> **User story A.2:** As an urban planner, I want to automatically detect individual trees from a LiDAR point cloud in an urban area, so I can monitor growth and foresee pruning work.

> **User story A.3:** As an urban planner, I would like to update my existing 3D city model based on analysing recent LiDAR point clouds.

> **User story A.4:** As an urban planner, I want to provide architects and other urban planners online access to the 3D city model using a simple lightweight web client embedded in any kind of web browser, so that they are able to integrate their projects into the model and share it with decision makers and citizens for communication and project assessment purposes.

Specific focus is put on the process of updating a 3D city model based on LiDAR point clouds (user story A.3). In order to complete this task, the urban planner typically performs the following operations:

i. Remove non-static objects (such as cars, rubbish bins, bikes, or people) from the point cloud.

ii. Characterize changes of static objects (such as trees, bus stops, or façade elements).

iii. Include new or exclude existing classified objects (e.g. roof tops or antennas).

iv. Use the result to update the city model (i.e. apply the changes to the existing model).

The urban planner typically visualises the results in 3D to assess correctness and overall quality. Such a 3D visualisation can be presented to decision makers, for example, if changes in the city such as new constructions should be discussed. This is reflected in user story A.4 and also described in our previous work (Dambruch & Krämer, 2014). We do not analyse A.4 in detail in the following sections because it contains many technical elements. We only take into account that the users wish to visualise the processing results at the end.

4.4.1 Vocabulary/Taxonomy

According to *step 3* of the DSL modelling method, the user stories from the previous section now have to be analysed to find subjects and objects that can later be used as classes in the domain model. Verbs and adjectives are also important. They will become actions and parameters in the Domain-Specific Language in the end.

Tables 4.1, 4.2 and 4.3 depict the results of this analysis with regard to identified subjects/objects, verbs and adjectives. Please note that not all terms found in the user stories are actually relevant. For example, the verbs 'monitor' and 'foresee' as well as the objects 'growth' and 'pruning work' refer to something that happens *after* the urban planner has processed the data. They do not belong to the processing itself and hence do not appear in the Domain-Specific Language.

Also note, at certain points, the wording in the user stories is unclear. For example, the expression 'changes of static objects' is rather unspecific about what 'changes' actually are. The only possibility to resolve this issue is to ask the users. In our case, they said that for them 'changes' mean that objects are added to a dataset or removed from it. We therefore put the adjectives 'added' and 'removed' in the table.

We tried to identify the singular form of all subjects and objects. This helped us later to make the domain model more consistent. In the case of 'people' we chose 'person'.

topographic object	cable network	street edge
urban furniture	traffic light	image
LiDAR point cloud	mobile mapping system	topographic city map
tree	urban area	growth
pruning work	3D city model	object
car	rubbish bin	bike
person	bus stop	façade element
roof	antenna	

Table 4.1 Subjects and objects identified in the text analysis

capture	create	update
detect	monitor	foresee
remove	characterize	include
exclude	visualise	

Table 4.2 Verbs identified in the text analysis

non-static	static	recent
added	removed	

Table 4.3 Adjectives identified in the text analysis

4.4.2 Domain model

The next step in the modelling process is to create a domain model based on the text analysis and the subjects and objects found. Figure 4.2 shows the domain model for this example use case. `CityModel`, `PointCloud`, and `Image` are datasets containing topographic objects. Each `TopographicObject` is either a `StaticObject` (`Roof`, `Antenna`, `Facade`, `Tree`, etc.) or a `NonStaticObject` (`People`, `Bike`, `Car`, etc.).

The domain model helps to structure the heap of terms found in the text analysis and to differentiate between relevant and irrelevant words.

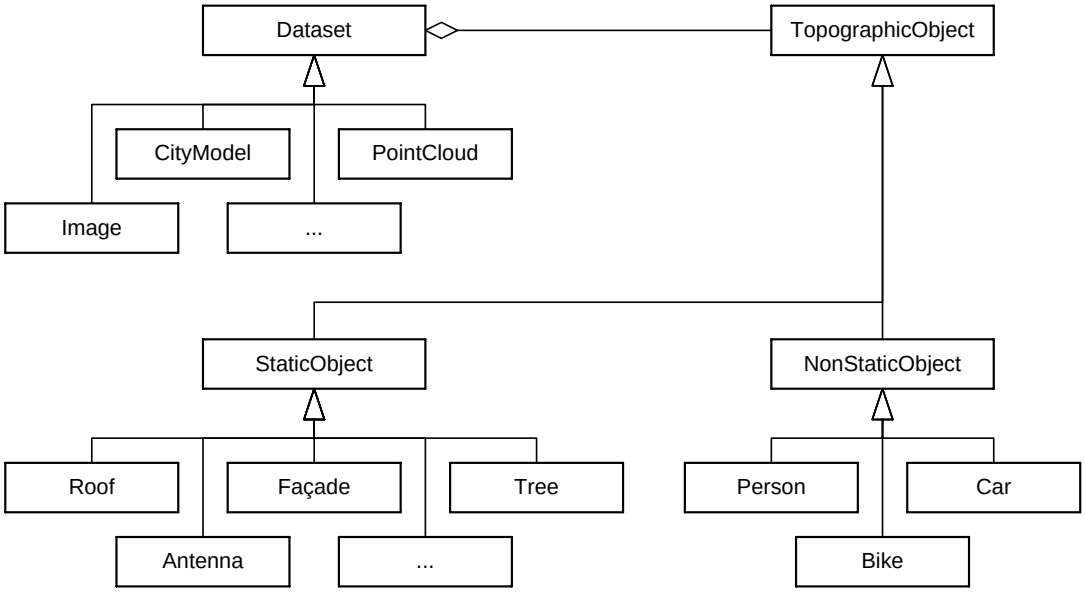


Figure 4.2 Domain model of the example use case A. For the sake of readability, some classes (datasets and static objects) have been left off.

4.4.3 Relevant verbs

In *step 5* of our modelling method we revise the list of verbs identified earlier and select those that should become actions in the Domain-Specific Language. We can split the verbs into two categories: one containing terms that relate to actions that the urban planner does (e.g. 'capture', 'monitor' and 'foresee') and another for commands the system should execute (e.g. 'remove', 'exclude' and 'visualise'). Applying this to all verbs from Table 4.2 results in the list of actions shown in Table 4.4.

detect	remove	characterize	include
exclude	visualise	update	

Table 4.4 Verbs that could become actions in the Domain-Specific Language

This list can further be simplified by merging similar terms. 'detect', 'characterise' and 'include' can be represented by the more generic verb 'select'. The final list is shown in table 4.5

select	exclude	update	visualise

Table 4.5 Final list of actions in the Domain-Specific Language

4.4.4 Sample DSL script

In *step 6* of our DSL modelling method we create sample scripts to be able to derive a generic grammar for the DSL.

In order to update the 3D city model and visualise the results (user story A.3), the following sample is proposed.

```
with recent PointCloud do
    exclude NonStaticObjects
    select added Trees and added FacadeElements
    update CityModel
end

with CityModel do
    exclude Antennas
    visualize
end
```

The script consists of two parts. In the first one a recently acquired point cloud dataset is processed. Non-static objects are removed and new trees and façade elements are detected. These new objects are added to the city model. In the next block all antennas are removed from the city model and the result is visualised on the screen.

The script is easy to read and shows exactly what processing steps are performed. The Domain-Specific Language proposed here makes use of terms from the domain model and from the results of the text analysis performed before. Domain users can therefore quickly learn the language and use it for their own needs.

Similar sample scripts can be created for the other two user stories A.1 and A.2.

4.4.5 DSL grammar

In order to make the Domain-Specific Language machine-readable, its grammar and syntax have to be formalised. A common way to do this is the specification of either a context-free grammar (CFG) using EBNF (Extended Backus–Naur Form). Alternatively, a PEG (Parsing Expression Grammar) can be used. One of the benefits of PEGs is that they can never be ambiguous. They are therefore very easy to define and are often not as complex as CFGs, not least because they do not require an additional tokenisation step. On the other hand, PEGs require more memory than CFGs, but for small languages such as DSLs this disadvantage can be neglected. The PEG for the sample DSL script presented in the previous section is as follows:

```
start = SP* statements SP*

statements = statement ( SP+ statement )*

statement = block / operation

block = with SP+ statements SP+ "end"

with = "with" SP+ dataset SP+ "do"

dataset = "recent" SP+ ID / ID

operation = "visualize" / "exclude" SP+ ID /
    "select" SP+ param SP+ ID ( SP+ "and" SP+ param SP+ ID )* /
    "update" SP+ dataset

param = "added"

ID = [_a-zA-Z][_a-zA-Z0-9]*

SP = [ \t\n\r]
```

The syntax used to specify the PEG here is the one of the tool PEG.js, an open-source parser generator written in JavaScript (Majda, 2016). Square brackets are used to define regular expressions. The plus character '+' means one or more occurrences whereas the asterisk '*' means zero or more occurrences. The slash character '/' is used to specify alternatives.

Note that a grammar for the complete example use case would be much larger. The PEG shown here can only be used to parse the example script from the previous section. For the sake of readability, additional grammar rules have been left off. We will create a complete grammar covering both example use cases in Section 4.5.

4.4.6 Reiteration

The final step of the modelling process is to review the Domain-Specific Language and to revise it if necessary. The language presented in this chapter already is a result of several iterations in which we worked together with domain users to create to a reasonable and sensible DSL that meets the users' expectations.

4.4.7 Rationale for the chosen syntax

The sample DSL presented before is based on the vocabulary from the domain model. Getting to the specific syntax was a matter of testing various alternatives and evaluating how they work in certain use cases. The following sample script was used as a starting point.

```
with PointCloud
exclude NonStaticObjects from it
select added Trees and added FacadeElements from it
add it to CityModel
```

In this sample script, the keyword `it` is used to refer to an object from the previous line or to refer to the result of the process performed in the previous line. This context-sensitive approach requires an intelligent parser that is able to clearly identify what `it` means in the respective context. While working with the domain users we noticed they had problems understanding such context-sensitive scripts. Instead, the following syntax was tested.

```
exclude NonStaticObjects from PointCloud
and select added Trees and added FacadeElements
and add to CityModel
```

In this case, individual processing steps are connected with the `and` keyword. While this approach leads to unambiguous scripts, they still can quickly become hardly readable. The longer the scripts get, the harder it is to understand them as the sentences become longer and longer. On the other hand, blocks enclosed by `with … do` and `end` (like they are used in Section 4.4.4) make clear which processing steps affect which data set.

4.5 Use case B: Land monitoring

In the last section we applied our DSL modelling method to the urban planning use case. We now do the same for the other use case dealing with land monitoring (see Section 1.8.2). In doing this, we try to align the DSL grammar with the one from Section 4.4.5. This allows us to create a Domain-Specific Language that is flexible enough to cover both use cases.

Again, we start with analysing the user stories.

> **User story B.1:** As an hydrologist or a geo-morphologist supporting decision makers in civil protection, I want to analyse data measured during critical events to prepare better prediction and monitoring of floods and landslides.

> **User story B.2:** As an hydrologist, I want to study the evolution of measured precipitation data as well as slope deformation from optical images, compute parameters to produce high-quality input for hydrological and mechanical modelling and simulation, and compare the results to reference measurements obtained for flooding events and landslides.

User story B.1 describes the overall goal of this use case. B.2, on the other hand, contains three individual steps to achieve this goal:

1. Study the evolution of precipitation data as well as slope deformation.

2. Compute parameters to produce high-quality input for hydrological and mechanical modelling and simulation.

3. Compare the results to reference measurements.

The first step is something that the user does outside the system. It can be ignored. Step 2, however, is rather complex and actually consists of the following partial steps:

i. Resample the available terrain data (point cloud) to match a given density.

ii. Remove outliers from the point cloud.

iii. Split the point cloud into areas defined by drainage boundaries.

iv. Reorder points according to their "relevance"—i.e. how much they contribute to the appearance of the terrain—and store them together with the drainage basins hierarchy.

For the final step 3 the following additional partial steps have to be performed:

v. Extract a single level of detail (i.e. a single resolution) from the reordered points.

vi. Perform a constrained triangulation to create a mesh from the point cloud preserving constraints such as boundaries and feature lines.

4.5.1 Vocabulary/Taxonomy

Similar to the previous use case, we now analyse the user stories and extract relevant subjects/objects, verbs and adjectives. The results are shown in Tables 4.6, 4.7, and 4.8.

Again, we list the singular form of all subjects and objects. We identified the base form of verbal nouns such as 'prediction' (to 'predict') and 'monitoring' (to 'monitor'), but also 'modelling' (to 'model') and 'simulation' (to 'simulate'), and put them in the list of verbs instead of objects.

event	flood	landslide
evolution	precipitation data	slope deformation
optical image	parameter	input
reference measurement	outlier	terrain data
point cloud	density	area
drainage boundary	relevance	terrain appearance
drainage basin	hierarchy	level of detail
resolution	triangulation	mesh
constraint	feature line	

Table 4.6 Subjects and objects identified in the text analysis

analyse	prepare	predict
monitor	study	compute
model	simulate	compare
remove	resample	split
reorder	store	extract
perform	create	

Table 4.7 Verbs identified in the text analysis

critical	high-quality	hydrological
mechanical	constrained	

Table 4.8 Adjectives identified in the text analysis

4.5.2 Domain model

According to our DSL modelling method, we now build the domain model based on the results from the previous section.

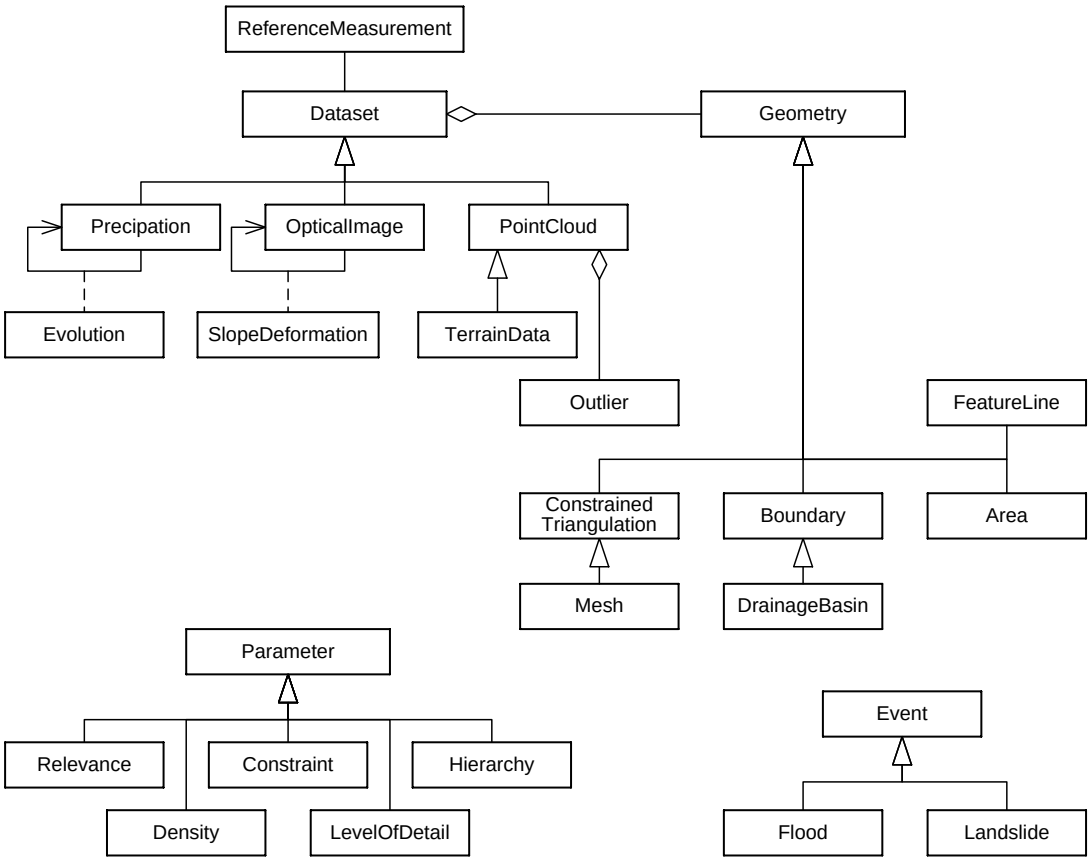


Figure 4.3 Domain model of the example use case B

We make the following assumptions:

- We do not differentiate between input and output. Similar to the domain model from Section 4.4.2 we use the term 'dataset' instead.
- We do not include 'terrain appearance' in the diagram. It just appears in a sub-clause to describe the term 'relevance'.
- The terms 'area', 'feature line' and 'drainage boundary' refer to geometries. We therefore add the term 'geometry' and reduce 'drainage boundary' to 'boundary'.
- A 'mesh' is defined by a 'triangulation'. Both are geometries.
- In the context of the use case a 'single resolution' is a 'level of detail'. We only include the latter.
- In Section 4.4.2 we merged objects and adjectives. Here, this is only necessary for 'constrained' and 'triangulation'. The other adjectives are either rating ('critical' and 'high-quality') or refer to verbal nouns ('hydrological' and 'mechanical').

4.5.3 Relevant verbs

Similar to the first use case, we now identify verbs that should become actions in the Domain-Specific Language. Again, we remove verbs that refer to activities the user performs outside the system (e.g. 'analyse' or 'prepare'). The verbs 'compute', 'perform' and 'create' describe similar activities. We therefore only include 'create'. The result is shown in table 4.9.

create	remove	resample	split
reorder	store	extract	

Table 4.9 Verbs that could become actions in the Domain-Specific Language

4.5.4 Sample DSL script

According to our DSL modelling method (step 6) we now create sample scripts. We try to reuse as much of the grammar from the previous use case as possible and propose the following script covering the first four sub-steps from user story B.2:

```
with PointCloud do
    remove Outliers
    resample using density: 10
    split with DrainageBoundaries as areas

    for each areas do
        reorder using method: "relevance"
        store
    end
end
```

The script operates on a point cloud. It first removes all outliers and resamples the dataset according to a given density. It then splits the resampled point cloud along given drainage boundaries. The result is a list of smaller point clouds named 'areas'. The script iterates over all areas and reorders the points according to their relevance. It stores each result to the Cloud.

The final two sub-steps from user story B.2 can be covered by the following script:

```
with Area do
    extract LevelOfDetail using lod: 10
    create ConstrainedTriangulation
end
```

Compared to the DSL from use case A this sample script introduces the following new keywords and language constructs:

- The keyword 'using' can be used to specify a value of a named parameter in the form 'name: value'. Parameter values can be numbers or strings (in double quotes).
- In the first use case we used the keyword 'with' to apply a number of operations to a dataset. We put these operations in a 'do … end' block. The new sample DSL script enables using 'with' in a single operation too.
- The keyword 'as' can be used to name the result of an operation.
- The new construct 'for each' allows for applying a number of operations to a list. The operations are specified in a 'do … end' block.

Creating the domain model in Section 4.5.2 was particularly useful in this case because it helped us differentiate between datasets (which are specified by 'with <dataset>') and parameters (specified by 'using <name>: <value>').

With the new keywords we can generalise the grammar from the previous use case. We can replace 'select' by 'extract' and make use of 'using' to specify what type of objects we want to extract. The sample script from Section 4.4.4 now looks as follows:

```
with recent PointCloud do
    exclude NonStaticObjects
    extract StaticObjects using type: "Trees" and type: "FacadeElements"
    update CityModel
end

with CityModel do
    exclude Antennas
    visualize
end
```

4.5.5 Generic DSL grammar and properties

The final step of our DSL modelling process is reiteration. As mentioned before, the language presented in this chapter already is a result of several iterations. In these iterations we were able to identify a couple of generic language properties as well as additional elements. We also generalised elements such as the 'for' expression or the way in which operations can be applied.

The following list is an overview of the language properties as well as the additional and the generalised language elements.

Execution order. The DSL presented here is a declarative and functional language. The order in which statements are executed is not strictly sequential. In Chapter 3, *Processing* we have shown that our system executes processing services according to a dependency graph and may run two

or more processing services in parallel if possible and beneficial. The DSL presented here supports modelling of such a graph. Individual statements can depend on the results of their respective prior statement and with the use of names (see below) even any prior statement. Even though statements appear one after the other it does not mean they will be executed sequentially. In fact, the following three scripts mean the same thing:

```
remove Outliers with [PointCloudA] as cleanPointCloud
resample with [PointCloudB] using density: 10
split with cleanPointCloud and DrainageBoundaries as areas
```

```
resample with [PointCloudB] using density: 10
remove Outliers with [PointCloudA] as cleanPointCloud
split with cleanPointCloud and DrainageBoundaries as areas
```

```
remove Outliers with [PointCloudA] as cleanPointCloud
split with cleanPointCloud and DrainageBoundaries as areas
resample with [PointCloudB] using density: 10
```

Names. Our language offers a way to name the result of an operation using the 'as' keyword. Names can only be assigned once. Their meaning must not change during the course of a workflow. This means all names are actually constants or immutable variables. This allows us to avoid side-effects that would make workflow scheduling in a distributed environment too complex. The interpreter presented in Section 4.6 performs semantic validation to prevent that a name is assigned more than once.

```
remove Outliers with PointCloud as newPointCloud
split with DrainageBoundaries as newPointCloud    // <- error!
```

For expression. In contrast to other (general-purpose) languages our 'for' expression is not a loop. There is no counter and no termination condition. The iterations are not necessarily executed sequentially. In fact, our 'for' expression can be better compared to a functional higher-order 'map' in which a set is mapped to another one by applying a given function f to all elements. Since our language does not allow for side effects, f can be applied to multiple elements in the set in parallel without causing conflicts.

Yield. A 'for' expression maps a set to a new set. Consequently, our language offers a way to make one 'for' expression depend on the results of another. We introduce the 'yield' keyword allowing users to explicitly declare the elements of which the new set should consist. This keyword is optional. If it is left off, the results of the last operation inside the 'for' expression will be collected into the new set. The following scripts therefore mean the same:

```
for each PointCloud do
    reorder using method: "relevance" as reorderedPointCloud
    yield reorderedPointCloud
end as setOfReorderedPointClouds

for each setOfReorderedPointClouds do
    resample using density: 10
end

// means the same as:

for each PointCloud do
    reorder using method: "relevance"
end as setOfReorderedPointClouds

for each setOfReorderedPointClouds do
    resample using density: 10
end
```

Note that in order to be able to iterate over the result of a 'for' expression we have to give it a name (in this case 'setOfReorderedPointClouds').

Generic 'apply'. In Section 4.6 we will show that our interpreter maps language terms to processing services. This allows for a high-level workflow description without requiring knowledge of the underlying execution system and the processing services that are actually applied. However, during the reiteration phase, some domain users requested to have more control over the processing services and their parameters. Those users were more familiar with the actual algorithms available and wanted to apply them directly. We therefore introduced the keyword 'apply'. The following script applies a processing service called 'OutlierRemoval' to a point cloud:

```
apply OutlierClassificationInPointCloud with [PointCloud]
    using outlierFilteringK: 15 and outlierFilteringStddev: 3.0
```

This script means the same thing as the high-level expression we used in the previous examples where we relied on default values for 'outlierFilteringK' and 'outlierFilteringStddev':

```
remove Outliers with PointCloud
```

Placeholders. Similar to the generic 'apply' keyword, domain users requested to have more control over which dataset they apply the processing workflow to. They also wanted to be able to reuse scripts and apply them to multiple datasets. We therefore introduced means to specify placeholders in the workflow script. They represent a dataset and can be replaced (or filled in) by the user interface calling the workflow interpreter (see Section 4.6). Placeholders are specified in square brackets:

```
with [PointCloud] do
    ...
end
```

Appendix A, *Combined DSL grammar* shows the final PEG which comprises the grammar from Section 4.4.5, the elements from applying the modelling method to use case B, as well as the generalised language elements.

4.6 Interpreting the workflow DSL

In this section we describe how the Domain-Specific Language designed in this chapter can be interpreted and how the individual language elements can be mapped to processing services. In order to meet the quality attributes defined in Section 2.3.2, in particular modifiability, we propose a modular approach that enables us to change both the language and the processing services later on independently without affecting the other.

To this end, we implement a component called *Interpreter* that translates a workflow script to a machine-readable workflow model as described in Section 3.6.1. The interpreter first parses a workflow script into an abstract syntax tree (AST). It then traverses the AST, performs semantic checks and produces executable workflow actions for each visited node. The process is shown in Figure 4.4 and described in detail in the following sections.

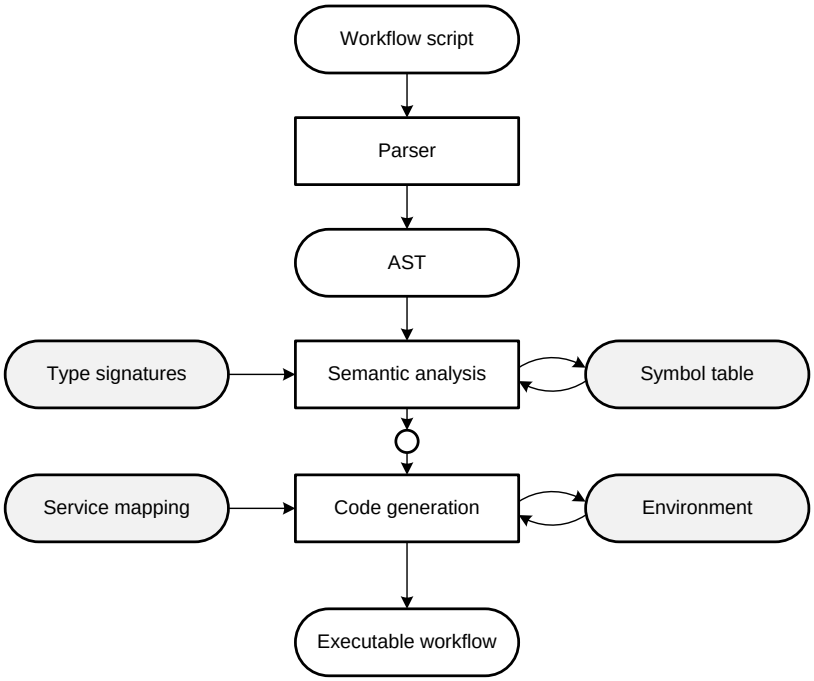


Figure 4.4 A workflow script is parsed to an Abstract Syntax Tree (AST) and then mapped to a machine-readable workflow model

4.6.1 Parsing and traversing

The DSL grammar shown in appendix A, *Combined DSL grammar* can be used to generate a parser with the open-source tool PEG.js. This parser produces an Abstract Syntax Tree (AST) that represents the individual tokens in the workflow script.

For example, consider the following workflow:

```
for each [PointCloud] do
    resample using density: 10
end
```

This workflow is parsed to the AST depicted in Figure 4.5. The root node in the AST represents the whole workflow. The other nodes denote the individual expressions and language elements. Each node has a number of properties that are either literals or link to other nodes (sub-expressions).

The AST is traversed twice using DFS (Depth-First Search). The first time the interpreter performs semantic checks and builds a symbol table (see Section 4.6.2). The second time it translates the AST nodes into executable workflow actions (see Section 4.6.3).

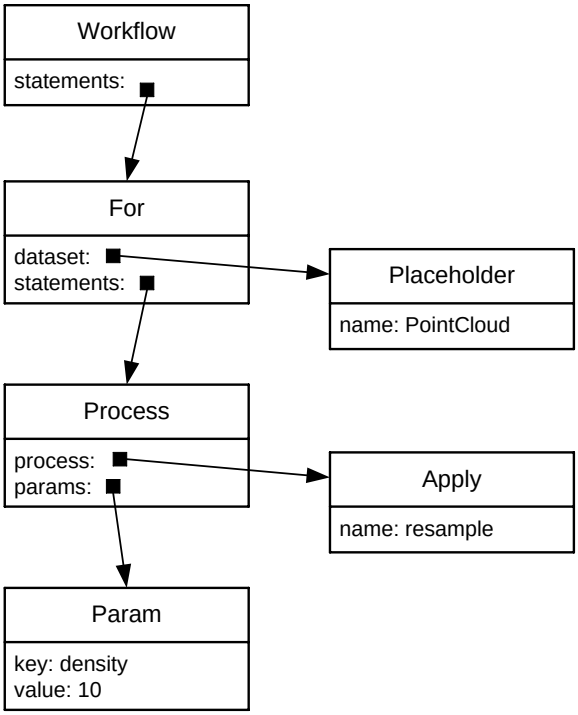


Figure 4.5 An example Abstract Syntax Tree

4.6.2 Semantic analysis

In the semantic analysis phase the interpreter traverses the AST and checks the semantics of the workflow script. It makes use of type signatures defining inputs and outputs of the individual operations. The set of signatures is mainly derived from the service metadata described in Section 3.6.2 but may also contain custom elements for language constructs that cannot be mapped one-to-one to a processing service.

The interpreter uses the type information to perform the following validations:

- A statement must be mappable to one or more processing services (see Section 4.6.3).
- A statement must contain values for all mandatory parameters of the processing services it maps to. A missing value may be filled in from the implicit context—either by using the dataset from an enclosing 'with' or 'for' block, or the result from the previous statement.
- A statement must not contain additional parameters not defined in the type signatures.
- The types of all parameters must be correct (according to the type signatures).
- Names must be declared before use.
- Names must not be assigned more than once.

During semantic analysis the interpreter also creates symbol tables. This data structure keeps information about names such as their type or the location of declaration. The interpreter uses these symbol tables to check for the existence of names and to prevent duplicate name assignments.

The symbol tables are kept in a stack. At the beginning, the top of the stack is the global symbol table (the one that contains declarations made on the top level of the script). Whenever the interpreter visits a block node ('with' or 'for') in the AST, it puts a new symbol table on top of the stack and removes it again when it has completely visited the node and all its children. In order to look up a name in the stack—i.e. to obtain its type or to check if it has been declared before—the interpreter starts with the symbol table on the top and then continues to the bottom until it either finds the name or reaches the end of the stack.

4.6.3 Code generation

In order to create a workflow structure that can be executed by the JobManager (see Chapter 3, *Processing*), the interpreter traverses the AST a second time. It makes use of a set of pre-defined rules that map AST nodes to processing services, service parameters and datasets. To illustrate the different types of mappings, Figure 4.6 shows examples that appear in use case A.

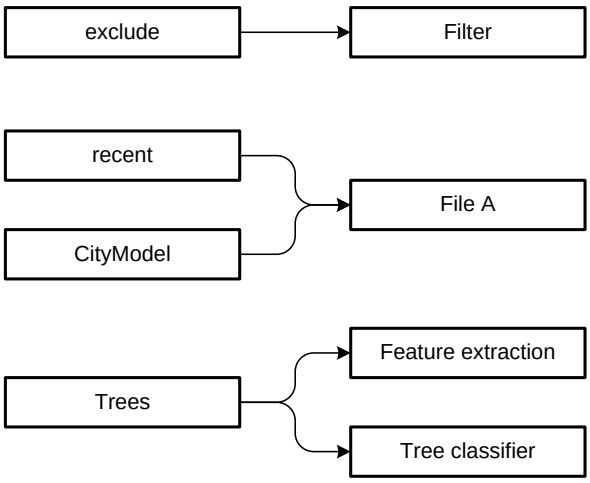


Figure 4.6 Nodes in the Abstract Syntax Tree are mapped to data sets and processing services

- **One-to-one mapping.** If a term such as 'exclude' appears in the AST, the interpreter maps it to exactly one processing service—in this case a filter removing objects that should be excluded.

- **Many-to-one mapping.** The terms 'recent' and 'CityModel', for example, are mapped to a specific dataset (or file) which represents the latest version of the 3D city model kept in the distributed Cloud storage.

- **One-to-many mapping.** Terms such as 'Trees' may need to be mapped to parametrised processing services. For example, the processing service for feature extraction is implemented using machine learning algorithms. The term 'Trees' therefore needs to be mapped to both the feature extraction service and a pre-trained classifier for trees.

In addition, many-to-many relations can also happen although they do not appear in the example use cases.

The mapping rules are stored in a configuration file that can either be written in JSON or YAML. This allows the mapping to be modified without requiring the developer to recompile the interpreter.

An initial configuration file can be generated from the service metadata described in Section 3.6.2 by applying a one-to-one mapping for the processing services and all their parameters. This configuration can then be modified to customise the Domain-Specific Language.

By the use of mapping rules as it is proposed here, the code generator may be replaced without affecting the DSL, the parser, or the semantic analyser. This means that even if the back-end (i.e. the processing services or the mapping rules) are replaced or modified, the scripts written by the domain users stay the same. In particular, this ensures that domain knowledge the users put into the scripts remains valid for a long time even if the underlying Cloud infrastructure changes—e.g. if the infrastructure is transferred from one Cloud provider to another.

In addition, the mapping rules allow for creating a modular Domain-Specific Language that consists of a basic set of keywords as well as a number of elements that are dynamically mapped to processing services, parameters and datasets. The Domain-Specific Language we describe here is therefore not cast in stone but can be adapted to further use cases.

Similar to the semantic analysis phase the interpreter keeps a data structure in the code generation phase containing information about names and their current values. This data structure is called *environment*. The interpreter maintains a stack of environments analogous to the stack of symbol tables in the semantic analysis. Each environment contains key-value pairs for all names valid in the current context (i.e. scope). This data structure is necessary to infer parameters from the context, so the statements are correctly converted to linked actions in the machine-readable workflow model.

4.7 User interface

As described in Chapter 2, *Architecture*, our system contains an editor allowing users to define their processing workflows using our Domain-Specific Language. To create a prototype of such a workflow editor, we implemented a web-based application as shown in Figure 4.7.

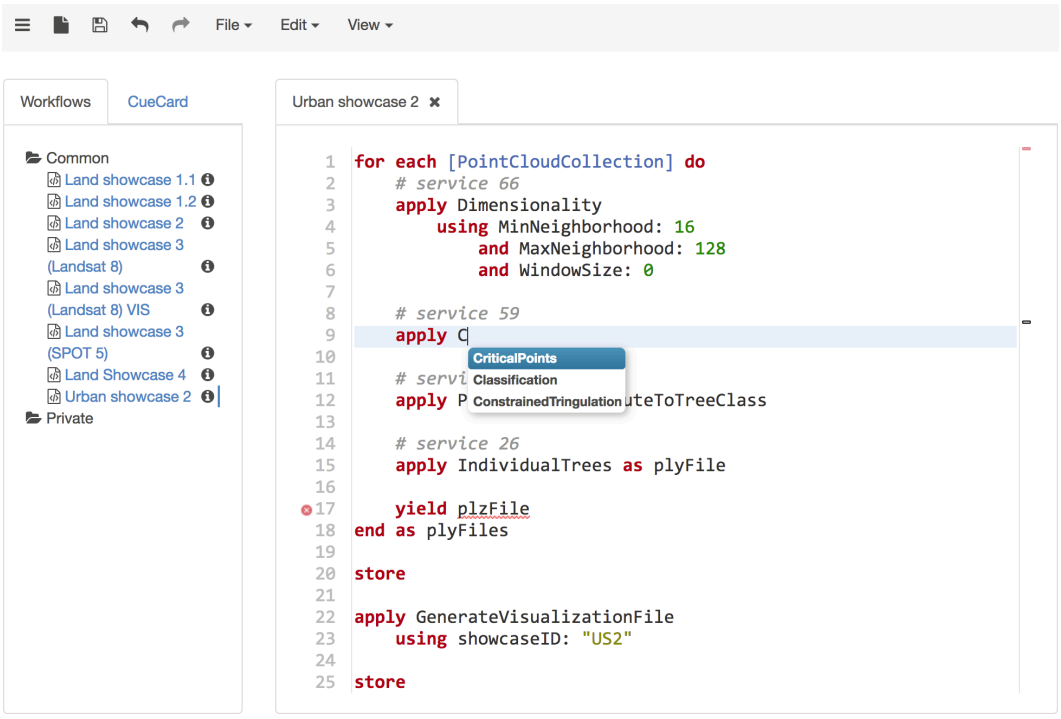


Figure 4.7 Screenshot of our web-based workflow editor

We designed the user interface according to the following principles: reuse existing standards and behaviours, and keep navigational elements at the necessary minimum. We evaluated some of the most popular Integrated Development Environments (IDEs) and editors such as Eclipse, IntelliJ IDEA, Visual Studio Code, and Sublime Text and selected elements relevant to our workflow editor. Our application consists of the following parts:

- The main area on the right where users can enter a workflow using the Domain-Specific Language
- The menu bar on the top with items to create a new workflow, save the current one, undo, redo, etc.
- A sidebar on the left containing
  - The list of all created workflows
  - A cue card showing hints while the user is typing

The editor allows users to design completely new workflows or to load and edit existing ones. Workflows are stored in a database. Users can decide whether they want to keep created workflows private or if they want to make them available to other users in the system.

The editor's main area offers a couple of useful features that support users in editing a workflow:

- *Syntax highlighting* helps identify the main elements of the DSL. The editor colourises individual tokens in the workflow according to their syntactical meaning. For example, keywords are displayed in red, numbers in green, comments in grey, and placeholders in blue.
- The *auto-completion feature* displays a list of suggestions while the user is typing. For example, if the user types a 'c' the editor will display a list of all keywords and processing services starting with this character. This feature speeds up the workflow definition and helps beginners to learn the language faster.
- *Error reporting* highlights issues in the workflow. The editor automatically compiles and validates the code in the background. Invalid statements (e.g. typos, missing processing services, invalid parameters) are underlined in red and an icon is displayed next to the number of the line containing the invalid statement. The user can hover the mouse over the icon to get a tooltip with the complete error message.

The sidebar on the left either displays a list of all workflows stored in the database and accessible to the user, or a cue card showing hints while the user is typing. The cue card's contents change dynamically depending on where the user has placed the cursor in the editor. For example, if the cursor is on a keyword the cue card will show a description of this keyword. If the user has typed an 'a' the cue card will display all keywords and processing services starting with this character including their description. If the user has selected the name of a processing service, the cue card will display the service parameters including descriptions and information about whether these parameters are mandatory or optional as well as their default values. Figure 4.8 depicts two example cue cards—one showing a processing service named 'Dimensionality' and its parameters, and another one showing keywords the user may insert at the current cursor position including a short description.

We implemented the workflow editor using AngularJS (Google, 2017). This framework supports the MVC (Model-View-Controller) pattern and allows for modularising the web application so that new components can be added later without affecting existing ones.

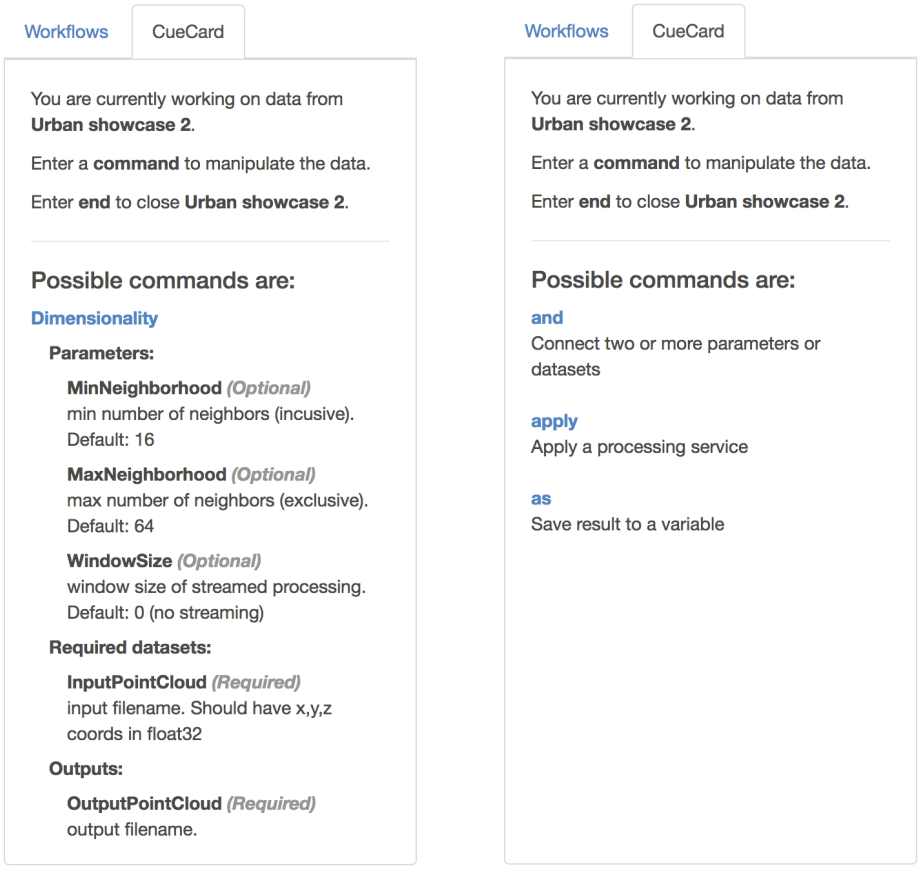


Figure 4.8 Two cue cards showing information about a processing service named 'Dimensionality' (left), and a list of keywords (including their description) which the user could enter at the current cursor position (right)

4.8 Summary

In this chapter we described a user interface for the processing of large geospatial data in the Cloud. The interface is facilitated by a Domain-Specific Language. In order to create a language that uses domain vocabulary, we introduced a novel method for DSL modelling. This method makes the language easy to understand and learn, and allows domain users to write their own processing scripts without a deep knowledge of the underlying Cloud infrastructure. In particular, the users do not have to care about on what specific hardware the processing is executed or what algorithms are exactly used and with what parameters they are called. The Domain-Specific Language offers just as much control as the users need. The rest is hidden in mapping rules that control how the language interpreter translates terms in a script to actual calls of processing services in the Cloud.

This chapter specifically focused on two use cases related to urban planning and land monitoring, but both the DSL modelling method and the approach to map language terms to processing services are independent of any application domain and can be applied to other areas as well. Due to the configurable mapping approach, our language is very modular. It can easily be extended by registering new processing services and amending the mapping rules.

The designed language contains generic constructs such as the 'apply' keyword which allows users to control exactly what services should be executed with which parameters, in contrast to

the high-level language constructs which provide a better usability by hiding details. Generic keywords and high-level domain-specific constructs can be intermixed in the same workflow. This allows domain users to select the right balance between usability and flexibility, depending on their personal experience.

Basically, the approach presented in this chapter is intended to close the gap between the processing of geospatial data in the Cloud and the end users who are GIS experts, but typically not experts in Computer Science, and in particular not in Cloud Computing or Big Data. In the future, Cloud technology will be used more and more often in the geospatial domain. Previous work has shown that this development has many benefits for all stakeholders (Khan et al., 2013; Krämer et al., 2013), but nonetheless a user interface that is easy to understand (like one that is based on a Domain-Specific Language) may be key to its success.

5 Evaluation

In this chapter we present the results from performing a qualitative and quantitative evaluation of our architecture and our implementation respectively. We show that our system meets the stakeholder requirements and the quality attributes specified in Chapter 2, *Architecture*. To this end, we first define specific evaluation scenarios and apply them to real-world data sets from our use cases introduced in Section 1.8. This enables us to validate whether the system actually has the specified quality attributes or not. After this, we discuss stakeholder requirements and how our system satisfies them. We also validate that the overall objectives of our thesis and the general requirements from our two user groups from the problem statement are met. The chapter concludes with a summary of the evaluation results.

5.1 Environment

In order to perform the quantitative evaluation, we deployed our system to a Cloud environment. At the Fraunhofer Institute for Computer Graphics Research IGD, Darmstadt, Germany we had access to an OpenStack Cloud (OpenStack Foundation, 2017) with 3 physical controller nodes, 13 compute nodes, and 6 storage nodes with the following specifications:

Controller and compute nodes:

CPU: 2 × Intel® Xeon® E5-2660v4 2.0GHz 14 Core LGA 2011

RAM: 8 × DIMM DDR4-2400 32GB ECC REG

HDD: 2 × Samsung SSD SM863 480GB

Network: 4 × 10GBit/s Ethernet

Storage nodes:

CPU: 2 × Intel® Xeon® E5-2620v4 2.1GHz 8 Core LGA 2011-3

RAM: 8 × DIMM DDR4-2400 32GB ECC REG DR

HDD: 4 × Samsung SSD SM863 960GB, 14 × Hitachi 3TB SATA 64MB 7200rpm

Network: 2 × 1GBit/s Ethernet

On this infrastructure we set up 24 virtual machines. We used six of them to host our core system services except the processing connector—i.e. the JobManager, workflow editor, data access service, data catalogue and service catalogue. We also deployed monitoring and logging services to these machines (see Section 5.1.1). We used the other 18 virtual machines as compute nodes with the processing services and the processing connector installed. They had the following specifications:

CPU:	2 cores
RAM:	8 GiB
OS:	Linux, Ubuntu 16.04 LTS 64-bit
HDD:	2 × 160 GiB

Each virtual machine had two virtual hard drives. One drive was used for the operating system, applications (including processing services), and temporary files. The second drive was reserved for the geospatial data we processed. We used GlusterFS to connect all virtual machines and to create a distributed file system with a total capacity of 2.81 TiB (160 GiB × 18). In order to introduce redundancy for fault tolerance and to improve read performance, we configured a replication factor of 3 in GlusterFS. Every file we uploaded to the distributed file system was therefore copied to three different virtual machines. We did not enable striping, so a single file was always stored completely on one node and not split into multiple fragments distributed over several nodes.

We deployed and configured the whole system using the IT automation tool Ansible (Red Hat, 2017). The process is further described in Section 5.3.6.

5.1.1 Monitoring and logging

In order to implement distributed logging and monitoring as described in Section 2.12, we set up an infrastructure as depicted in Figure 5.1.

We deployed an Elastic Stack (formerly known as ELK stack) consisting of Logstash, Elasticsearch and Kibana (Elasticsearch BV, 2017). Logstash filtered and transformed incoming log mes-

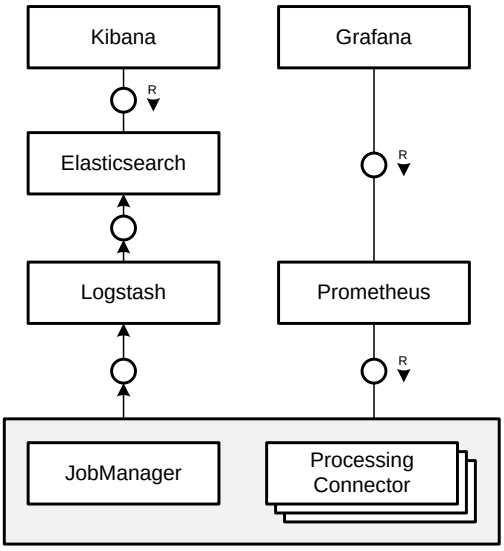


Figure 5.1 Distributed logging and monitoring

sages from our services and stored them in Elasticsearch where they were analysed and indexed. The web-based graphical user interface Kibana could then be used to query Elasticsearch and to search and aggregate large amounts of log messages by certain criteria. This allowed us to trace distributed workflow executions and to find bugs during development.

In addition to the Elastic Stack, we set up a monitoring chain based on Prometheus (Prometheus Community, 2016) and Grafana (Grafana Labs, 2017). Prometheus regularly polled our services for metrics. These metrics included information about the virtual machines our services ran on (CPU, memory, etc.) as well as application-specific values (e.g. the number of process chains currently being executed). We used Grafana to aggregate and visualise the metrics and to create a dashboard with which we could monitor the execution of workflows. The graphs we put into this chapter are screenshots from Grafana.

5.2 Use cases

In order to perform the quantitative evaluation of our system implementation under realistic conditions, we executed workflows from our use cases defined in Chapter 1, *Introduction* several times under varying conditions. The following two sections describe these workflows and the data sets we applied them to.

5.2.1 Use case A: Urban planning

The main tasks in our urban planning use case are to keep cadastral data sets up to date and to monitor the growth of trees in the urban area. For both tasks, experts from the municipality or from a mapping authority analyse large LiDAR point clouds (Light Detection And Ranging) acquired by an LMMS (Laser Mobile Mapping System) and try to extract objects such as urban furniture, traffic lights, façades and trees.

In our evaluation we focused on user story A.2, the extraction of trees from LiDAR data in order to monitor their growth and to foresee pruning works. In order to automate this task, we worked together with domain experts from the national mapping agency of France (Institut Géographique National IGN) and created the following workflow in our Domain-Specific Language:

```
for each [PointCloud] do
    apply Dimensionality
        using MinNeighborhood: 64
            and MaxNeighborhood: 256
            and WindowSize: 0

    apply Classification

    apply PointCloudAttributeToTreeClass

    apply IndividualTrees

    store
end
```

The workflow operates on a number of point cloud tiles acquired by an LMMS. It applies four processing services and then stores the result for each tile to the distributed file system.

Note that we used the generic `apply` keyword (see Section 4.5.5) to call individual processing services instead of high-level expressions such as `select Trees`. This allowed us to have a more fine-grained control over which services are called and to monitor their execution more precisely in

our evaluation. The advantage of this will become clearer in use case B in Section 5.2.2, where we correlate the recorded metrics to service calls in the workflow. Nevertheless, high-level expressions would have yielded the same results.

The workflow contains four processing services created by software developers from different institutions—i.e. University College London UCL, Delft University of Technology, and the French national mapping agency IGN. The 'Dimensionality' service determines the shape of the neighbourhood for each point. It calculates three dimensionality features on spherical neighbourhoods at various radius sizes and determines if a point and its neighbours lie on an edge, a surface or a volume (Demantké, Mallet, David, & Vallet, 2011). The service can be applied to 3D point clouds from airborne, terrestrial and mobile mapping systems and requires no a priori knowledge about the distribution of vertices.

The calculated dimensionality provides significant hints for the 'Classification' service. This service identifies tree crowns in point clouds by applying a Random Forest supervised classifier (Breiman, 2001). The service adopts a simplified version of the pipeline from Weinmann, Jutzi, & Mallet (2014). It consists of three steps: a feature selection that reduces the number of features to classify, the actual classification, and a final regularisation step which produces homogeneous labels by selecting an 80% majority label in a neighbourhood (Böhm et al., 2016).

The third service is called 'PointCloudAttributeToTreeClass'. This service analyses the classified point cloud and divides the points into two classes: tree and non-tree. The decision whether a point belongs to a tree or not is based on a scattering label that is added to the point cloud by the 'Classification' service. It is assumed that for tree points the scattering value is higher than for non-tree points. The output of 'PointCloudAttributeToTreeClass' is a new point cloud containing only tree points.

The last service 'IndividualTrees' divides the remaining set of points into individual objects. It applies a number of heuristics to create minimal non-overlapping bounding boxes around points belonging to the same tree. The final result is a set of points labelled with unique numbers. Points with the same number belong to the same tree.

Data set

We applied the workflow to a large data set acquired by an LMMS in the city of Toulouse, France. The data set had the following characteristics:

Number of point cloud tiles	529
Number of vertices per tile	3,000,000 (some tiles at the borders of the data set had less points)
Total number of vertices	1,580,600,752 (1.58 billion)
Acquisition time	1 hour and 53 minutes
Total volume	120.63 GiB

Collected metrics

The following figures show metrics we collected while applying the workflow to our example data set. The whole process took *1 hour and 51 minutes*. Figure 5.2 shows the CPU usage on each of the 18 compute nodes (18 coloured lines) and Figure 5.3 the free memory per node. Figures 5.4 and 5.5 show the network usage and the disk usage respectively, summed up over all of the 18 compute nodes.

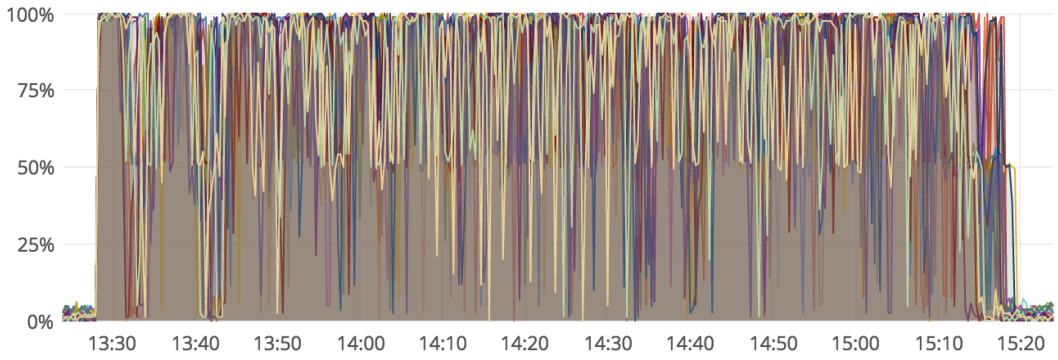


Figure 5.2 Use case A: CPU usage

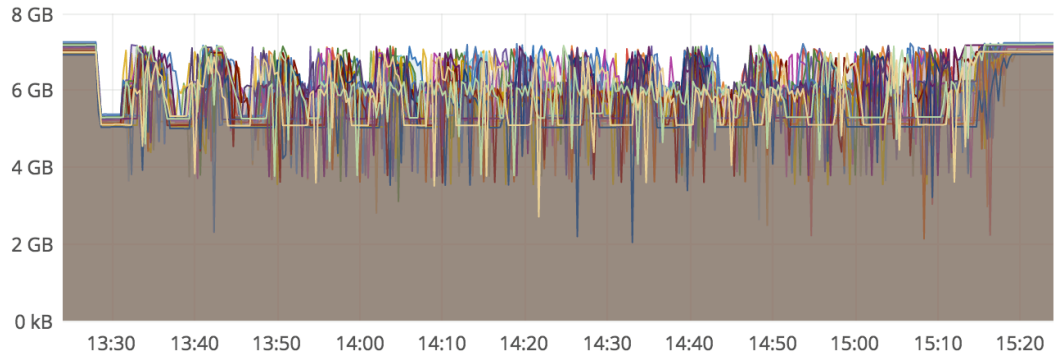


Figure 5.3 Use case A: Memory usage

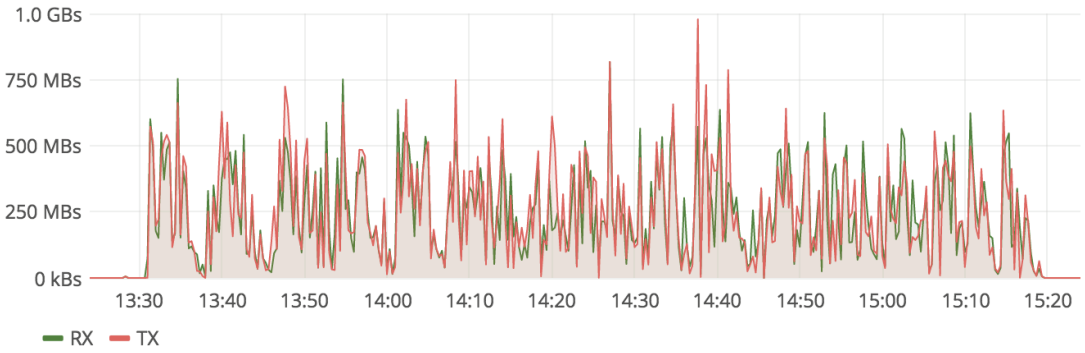


Figure 5.4 Use case A: Network RX/TX

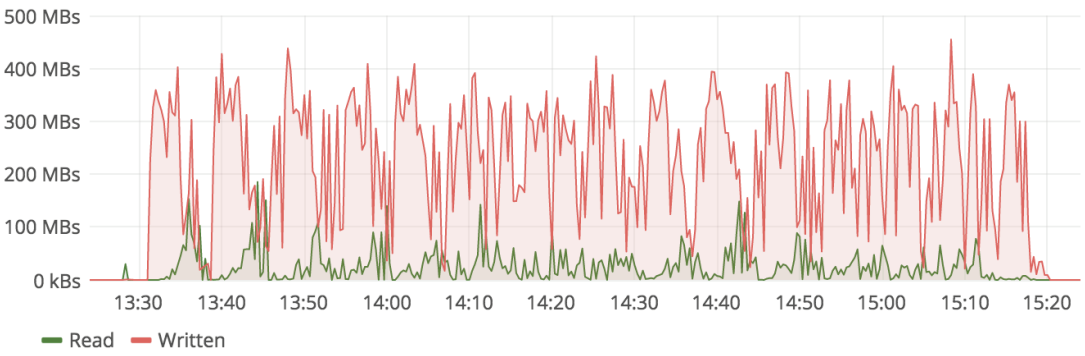


Figure 5.5 Use case A: Disk I/O

Since the processing services in this use case are single-threaded and each virtual machine had 2 CPU cores we set the scheduling queue size of the process chain manager to 2 (see Section 3.7.4). This means that the JobManager ran up to 2 processing services on each compute node. The graphs show that our system was able to parallelise the workflow execution and to make use of available computational power.

The green line in Figure 5.4 (network usage) indicates the number of bytes received per second and the red one the number of bytes transmitted. Both metrics are very close to each other, which indicates that the same number of bytes were received as transmitted. This is due to the fact that the processed data is completely stored on the distributed file system spanning over all compute nodes, and that it is only transferred between the nodes while there is almost no communication with external systems.

The number of bytes read from disk in Figure 5.5 is significantly lower than the number of bytes written. This is due to the fact that the individual processing services in this workflow produce large temporary files. For example, the 'Dimensionality' service reads a point cloud and generates a new one with the same number of points but additional attributes. The same is true for the 'Classification' service that adds labels and further attributes to each point.

5.2.2 Use case B: Land monitoring

In our second use case, expert users from the environmental department of the Liguria Region in Italy monitor the evolution of the region's terrain and try to derive knowledge that helps them prepare for future critical events such as landslides and floods. In this chapter we focus on the preparation of LiDAR data acquired by airborne laser scanning.

In order to study the terrain in the region, the expert users would like to select one or more drainage basins and visualise them as 3D surfaces. Our example data set (see below) is irregular. Each point cloud covers a strip of terrain (along the route of the airplane that carried the laser scanner). The length of these strips, the size of the point clouds as well as the covered area varies. Figure 5.6 shows that the strips and the drainage basins do not necessarily match. In order to generate a 3D surface, we first need to analyse the points in all strips and identify which drainage basin they belong to. In doing so, we create an index of points and basins. We then need to convert the points to 3D surfaces by applying a Delaunay triangulation. We generate multiple resolutions (levels of detail) of these surfaces from which the users can select, depending on their requirements for the visualisation.

Figure 5.6 An example of a point cloud strip that touches three drainage basins

The workflow we created together with users from the environmental department of the Liguria Region ("Regione Liguria") consists of two parts. The first part creates the index and reorders points according to their saliency. In the second part the expert users can select a specific level of detail and create a triangulation constrained to a certain drainage basin.

The workflow requires two input data sets: a collection of point clouds in the LAS format, as well as a Shapefile containing geometries representing the boundaries of each drainage basin in the region. The first part of the workflow is defined as follows in our Domain-Specific Language:

```
for each [PointCloud] do
    apply ResamplingOfPointCloud
        using resamplingResolution: 20

    apply OutlierClassificationInPointCloud
        using outlierFilteringK: 15
        and outlierFilteringStddev: 3.0

    apply VectorLayerPointCloudPartitioning
        with [boundaries]
end as results94

apply VLJsonMerger with results94 as merged94

for each merged94.outputMetaFolder do
    apply MultiresolutionTriangulation
end as results48

apply MTlasMerger with merged94.outputLasFolder

store
```

The workflow script contains two 'for' expressions. The first one iterates over the collection of point clouds and reduces their resolution with the 'ResamplingOfPointCloud' service. After that, it applies the 'OutlierClassificationInPointCloud' service which removes outliers—presumably measuring errors—from the resampled point clouds. This ensures that the resulting triangulation is smooth and resembles the actual terrain. The third service 'VectorLayerPointCloudPartitioning' splits an input point cloud along the drainage basin boundaries. The service creates multiple files, one for each basin containing points from the input data set. For example, the strip shown in Figure 5.6 would be divided into three files. The service also creates a JSON file for each generated point cloud containing metadata necessary for the services in the subsequent steps.

Since the point clouds are processed in parallel, multiple instances of 'VectorLayerPointCloud-Partitioning' may create different files for drainage basins covering more than one point cloud. The result of the first 'for' expression is hence a list of files. The service 'VLJsonMerger' accepts this list as input and merges metadata files belonging to the same basin to single files.

In a second 'for' expression the partitioned point clouds are then processed by the 'MultiresolutionTriangulation' service. This service orders points in a hierarchical manner according to their saliency—i.e. how much they contribute to the appearance of the terrain.

The final service 'MTlasMerger' merges point clouds belonging to the same basin to a single file. After the first part of the workflow has finished, users may choose to run a subsequent part that looks as follows:

```
for each [basins] do
    apply LodExtract
        using lodOfInterest: 10
        as extractedLod

    apply ConstrainedTringulation
        with extractedLod and [boundary]
end as results

store results
```

In this workflow script, users can define a certain level of detail they want to extract from the point clouds prepared in the first part. The 'LodExtract' service works on the points reordered by the 'MultiresolutionTriangulation' service and selects the most salient ones until it reaches a resolution that satisfies the given level of detail. After that, the 'ConstrainedTringulation' service performs a Delaunay triangulation and constrains the result to the drainage basin boundaries according to Shewchuk (1996). The whole process can be applied to multiple basins in parallel. The final results are stored in the distributed file system.

Note that in the following we focus on the first part of the workflow only. It covers a range of functionalities provided by our system and is therefore suitable to evaluate it. The second one, on the other hand, is typically only applied to a small portion of the data. It is not very complex and typically finishes in a short time. Performing tests with it would not reveal additional insights.

Data set

As described earlier, our workflow requires two input data sets. A collection of point cloud strips acquired by airborne laser scanning and a Shapefile containing the boundaries of all drainage basins in the Liguria region. In order to evaluate our system, we used a point cloud collection with the following properties:

Number of point cloud strips	684
Total number of points	17,345,032,915 (17.35 billion)
Total volume	451.16 GiB

In addition, we used a Shapefile with a total of 638 drainage basin boundaries. Figure 5.7 shows the areas covered by the point cloud strips. For a map of the drainage basin boundaries, we refer to Figure 1.2 (p. 11).

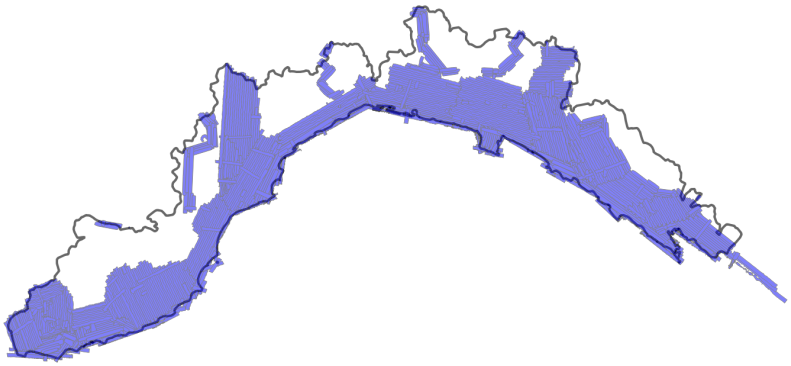


Figure 5.7 A map showing the areas covered by the point cloud strips in our example data set

Collected metrics

The metrics we collected while applying the land monitoring workflow to our example data set are shown in Figures 5.8 to 5.13. Similar to use case A, we collected CPU usage, memory usage, network traffic, and disk I/O. In addition, we recorded the number of available compute slots (which is the number of compute nodes multiplied by the size of the scheduling queue, see Figure 5.12) as well as the number of process chains the JobManager had to execute at a certain point in time (Figure 5.13). The whole process took *35 minutes and 49 seconds*.

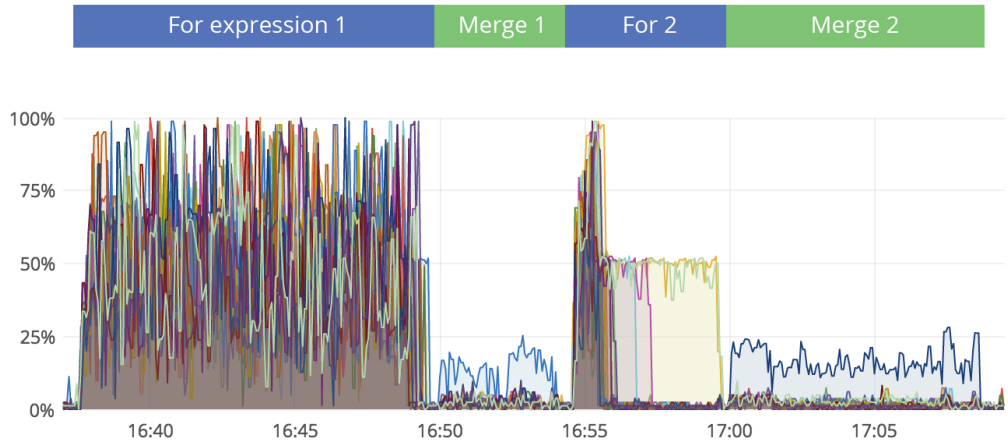


Figure 5.8 Use case B: CPU usage

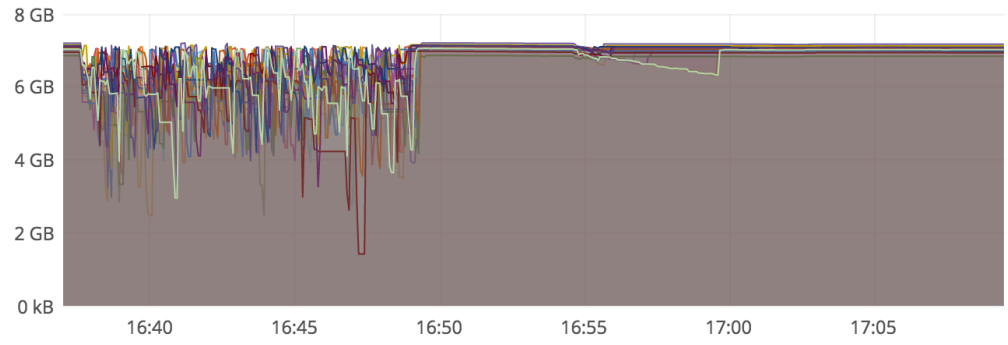


Figure 5.9 Use case B: Memory usage

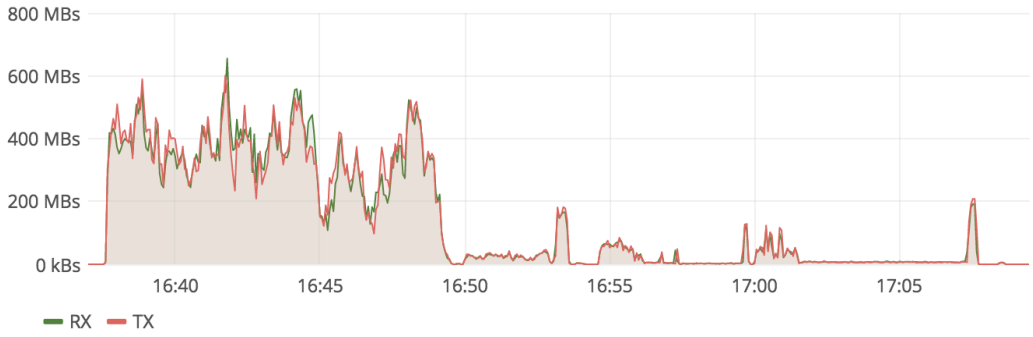


Figure 5.10 Use case B: Network RX/TX

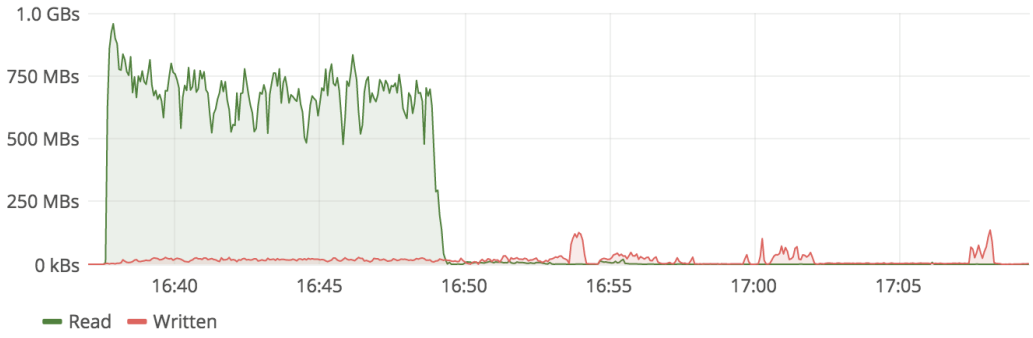


Figure 5.11 Use case B: Disk I/O

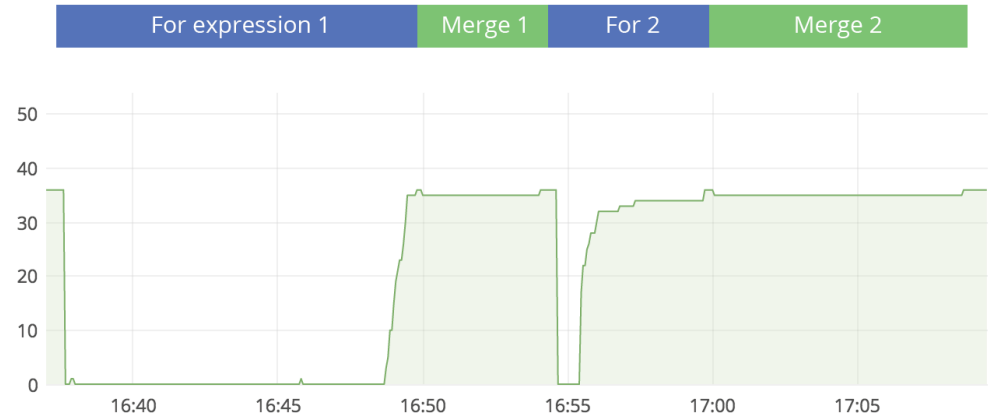


Figure 5.12 Use case B: Available compute slots

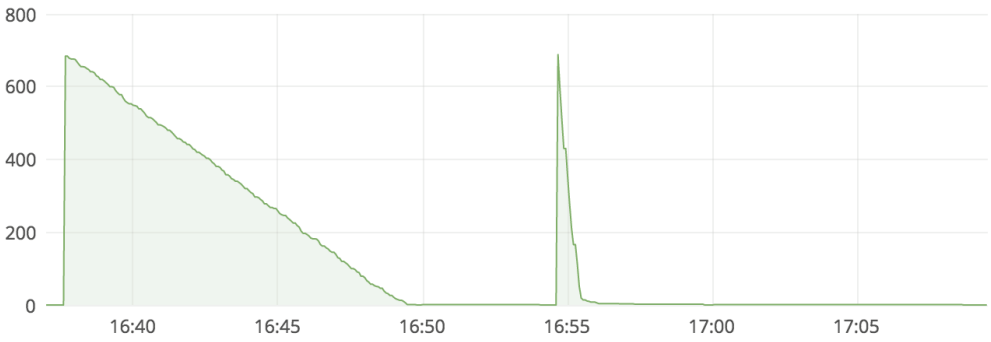


Figure 5.13 Use case B: Number of process chains to execute

In order to be able to better compare the figures and to correlate them with the workflow script, we created labels showing the four phases of the workflow: the first 'for' expression, the first merge step, the second 'for' expression, and the final merge step. The figures show that the CPU usage is very high in the first phase and that all compute nodes are almost completely used to their capacity. From Figure 5.13 we can see that the JobManager's rule system has created as many process chains as there are input point cloud strips and that they are constantly being processed. Figure 5.12 shows that these process chains are distributed to all compute nodes.

The CPU usage is not constantly at 100% in the first phase because there is also a lot of I/O at the same time. The point cloud strips are read from the distributed file system and transferred over the network to peer compute nodes. Note that the number of bytes transferred over the network per second is lower than the number of bytes read from disk. This is due to the fact that the JobManager is able to run processing services on nodes that actually contain the files to be processed. Since the JobManager is optimised for maximum throughput and our scheduling queue has a length of 2 only, this optimisation does not work all the time. There is a trade-off between the number of process chains the JobManager can execute in a certain amount of time and the overhead introduced by the amount of data transferred over the network. However, as we will show in Section 5.3.1, this has almost no effect on the overall performance of the workflow.

The second workflow phase represents the first merge operation. The figures show that only one virtual machine is occupied in this phase. CPU usage is not very high because the 'VLJsonMerger' service is single-threaded. In addition, it has to read a very large number of small files generated by the 'VectorLayerPointCloudPartitioning' service, which introduces latency not visible in the figures.

In the third phase the JobManager processes the second 'for' expression. Similar to the first one, there are a lot of process chains to be executed. They are distributed evenly to the compute

nodes. Except for a few, the individual process chains do not take very long to run. This is the reason why Figure 5.8 shows a spike in the CPU usage at the beginning of the phase and a longer period where only a few nodes are used to a maximum of 50%. The network traffic and disk I/O are relatively low in this phase because the original input files have already been reduced in size.

The final phase contains the last merge operation. Similar to 'VLJsonMerger' the 'MTlasMerger' service is single-threaded and can run on one compute node only. Again, the latency introduced by the fact that the service has to process a large number of very small files leads to a small CPU usage and almost negligible spikes in the figures for network traffic and disk I/O.

5.3 Quality attributes

In this section we report on the results of the qualitative evaluation of our system against the quality attributes we defined in Section 2.3.2. For each quality attribute, there is a general scenario describing an event, a system response, as well as measurable criteria to validate if our system actually responded as expected. For the evaluation we derived specific scenarios from the general ones which described how the system was expected to respond if it was applied to our use cases. We then performed the validation and recorded metrics. In this section we present the specific scenarios and discuss the validation results.

5.3.1 Performance

In Section 2.3.2 we specified two criteria to define what 'performance' means for our system (see Table 2.1 on page 33). The first criterion referred to the execution time of a workflow, and the second one to Cloud resource usage.

Execution time

In our urban use case we require that the time it takes to process a large LiDAR point cloud should be equal to or less than its acquisition time. We derived a specific scenario from this requirement in order to be able to evaluate if our system is fast enough (see Table 5.1).

Source	Users and GIS experts from municipalities, mapping authorities
Stimulus	Automatic tree detection within large point clouds acquired by an LMMS
Environment	Normal operation
Artefacts	Whole system
Response	The system processes the acquired data (i.e. it identifies individual trees) and offers the results for download
Response measure	The time needed to process the data is equal to or less than the acquisition time

Table 5.1 Performance Specific Scenario A - Time constraints

Comparing the acquisition time of our example LiDAR data set from Section 5.2.1 (*1 hour and 53 minutes*) with the processing time on 18 virtual machines shown in Figure 5.2 (*1 hour*

and 51 minutes) we can clearly state that our goal from use case A has been reached. It is indeed possible to process the data set in less time than it took to acquire it.

As we will show in Section 5.3.2 the workflow scales almost linearly. The performance could therefore be even more improved by adding further virtual compute nodes. Our end-users suggested to insert another processing service to the workflow which resamples the data to a lower resolution before it is processed by the other services (i.e. the 'ResamplingOfPointCloud' service, see Section 5.2.2). This would have reduced the processing time even further and—with a reasonable resampling factor—would have yielded almost the same results.

Resource usage

In order to evaluate whether our system makes best use of available resources, we defined a specific scenario as described in Table 5.2.

Source	Users and GIS experts
Stimulus	Processing of aerial point clouds for land monitoring
Environment	Normal operation
Artefacts	Whole system
Response	The system processes the data and offers the results for download
Response measure	While processing, the system makes best use of available resources *a)* All compute nodes are used to their full capacity in terms of CPU power *b)* The amount of data transferred over the network is minimised

Table 5.2 Performance Specific Scenario B - Resource usage

Response measure a) is satisfied by our system. Figure 5.8 (page 135) shows the CPU usage recorded during a workflow run of use case B on 18 compute nodes. For this run we set the size of the scheduling queue in our process chain manager to 2. This means that the JobManager always executed up to two processing services per compute node at the same time. Although the individual services were single-threaded, both CPU cores on each node were fully used.

In order to support a wider range of processing services and workflows, the size of the scheduling queue in the JobManager should be dynamic and vary depending on whether a single-threaded or a multi-threaded processing service is executed. The queue size should also depend on the number of CPU cores available on each compute node. For the experiments presented here, we assume that the processing services are single-threaded. Further, we assume that all compute nodes are homogeneous and have two CPU cores.

In order to validate response measure b), we ran the workflow for use case B another time on 18 compute nodes but disabled the data locality optimisation of the JobManager (see Section 3.7.3). This means that the processing services were executed on arbitrary nodes and not on those that host the respective input data. Figure 5.14 and Figure 5.15 show the network usage and the disk usage for this workflow run respectively. If we compare these two figures to the network usage and disk usage recorded in the first workflow run (see Figure 5.10 and 5.11 on page 135) we can state that while the disk usage has not changed, the amount of data transferred over the network is now much higher. This is due to the fact that the same amount of data had to be read from disk during the workflow run, but since the distribution of the processing services across the compute nodes was much more arbitrary, input data from the distributed file system had to be transferred between nodes more often.

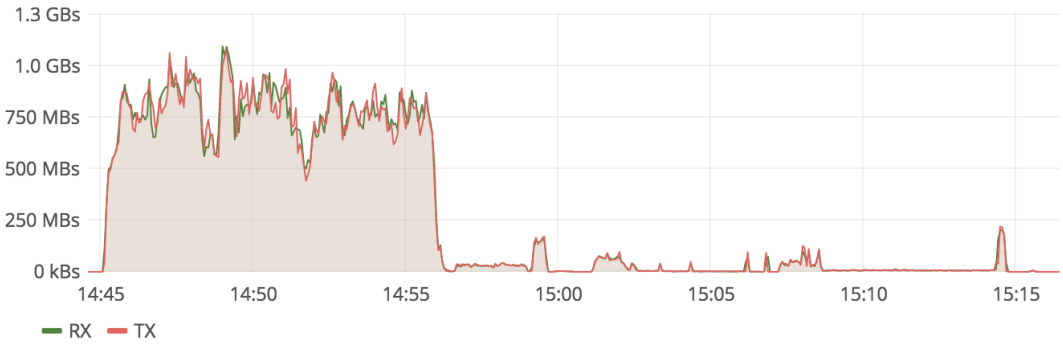


Figure 5.14 Use case B: Network usage with disabled data locality optimisation

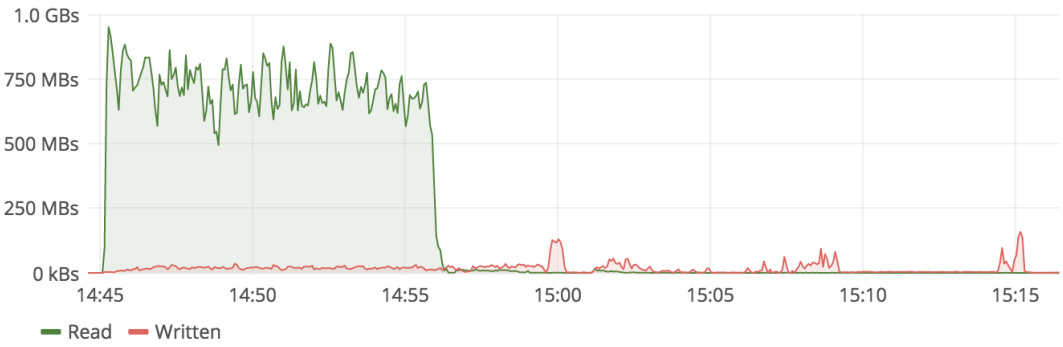


Figure 5.15 Use case B: Disk usage with disabled data locality optimisation

The workflow run with disabled data locality optimisation took *35 minutes and 1 second*. This is almost identical to the first workflow run from Section 5.2.2 which took *35 minutes and 49 seconds*. The difference is within the accuracy of measurement and subject to fluctuations that are to be expected in a dynamic Cloud environment. One would expect that the workflow execution without locality optimisation is much slower since more data has to be transferred over the network. However, in our case the network is much faster than the disk and hence the disk speed affects the execution time more than the amount of data transferred over the network.

Note that we performed our evaluation in a controlled and isolated environment. The results may vary on other Cloud infrastructures where the data is not stored in the same rack and probably even not located in the same data centre. In this case data would have to be transferred over a much slower network connection (possibly a WAN connection). Our graphs show that network traffic is reduced by our optimisation and that it is actually in effect. The impact on execution time will be more evident in set ups involving slower network connections.

5.3.2 Scalability

In Section 2.3.2 we specified a general scenario for the quality attribute *scalability* with three response measures:

- The system should continue to work under heavy load (multiple workflow executions at the same time)
- Workflows should be processed faster with more computational resources available
- The system should be able to handle arbitrary amounts of data

In this section we present specific scenarios for these three measures and report on the results of experiments we performed on the Cloud with our example workflows and data sets.

Multiple workflow executions

We first tested the execution of concurrent workflows over a defined period of time in order to evaluate how our system performs under high load. The specific scenario for this test is given in Table 5.3.

Source	Users, GIS experts, data providers
Stimulus	Execution of concurrent workflows
Environment	Overloaded operation
Artefacts	Whole system
Response	The system executes all submitted workflows
Response measure	1) All workflows complete as if they were executed one after the other under normal operation 2) The workflows are finished in reasonable time

Table 5.3 Scalability Specific Scenario A - Multiple workflow executions

We executed the workflow for our use case A twelve times on 18 virtual machines with a scheduling queue size of 2 in the process chain manager. We timed the executions so that we could simulate concurrent runs at the same time and high load over a longer period. The following table shows which workflow run was started at which point in time.

Run	#1	#2	#3	#4	#5	#6
Start	0h 00m 00s	0h 00m 05s	0h 01m 05s	2h 01m 05s	2h 31m 05s	2h 31m 10s
	#7	#8	#9	#10	#11	#12
	2h 31m 11s	2h 31m 12s	2h 31m 17s	2h 41m 17s	8h 41m 17s	11h 41m 17s

Figure 5.16 shows the number of process chains generated during the experiment. At the beginning there is a quick ramp up as three workflows were executed within 1 minute. The maximum number of process chains in the JobManager's queue was reached at about 2h 41m, after the first ten workflow runs had been started. From there on, there was a constant decline with two more spikes for workflow runs 11 and 12.

Figures 5.17 to 5.20 depict the Cloud resource usage over the whole period of time. The graphs show values very similar to the ones collected during our first run in Section 5.2.1. There are no specific peaks or other anomalies. This indicates that even under high load our system continued to operate as if it was executing only one workflow.

The whole experiment took *21 hours and 50 minutes*. Twelve sequential runs of our urban workflow would have taken *22 hours and 12 minutes* (1h 51m × 12). The concurrent runs were hence a bit faster. This is due to the fact that the JobManager could leverage parallelisation better, in particular at the end of each run where only a few process chains were left and a couple of compute nodes became already available again.

In summary, we can state that our system did not show any signs of overload during the experiment. It also stayed responsive all the time. Workflows could be started without any noticeable latency and metrics could continuously be retrieved from the JobManager and the compute nodes. This is primarily attributable to the fact that we designed the JobManager to be reactive (as described in Section 3.4) and that the individual microservices running in our system are isolated and do not affect each other's performance.

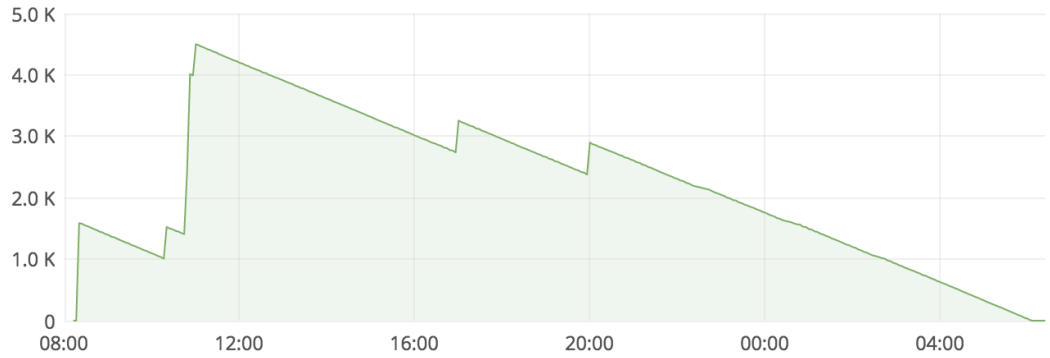


Figure 5.16 Concurrent executions of use case A: Number of process chains

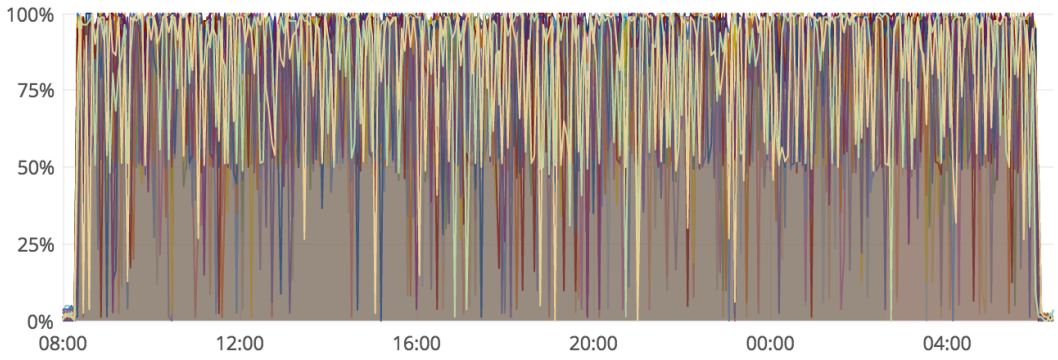


Figure 5.17 Concurrent executions of use case A: CPU usage

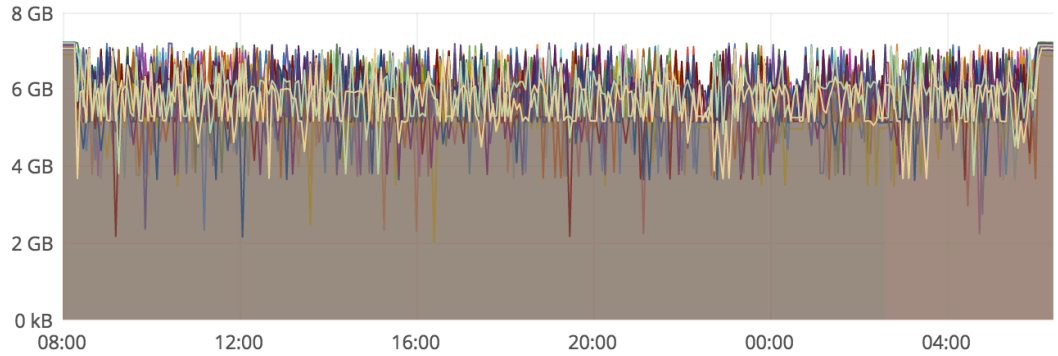


Figure 5.18 Concurrent executions of use case A: Memory usage

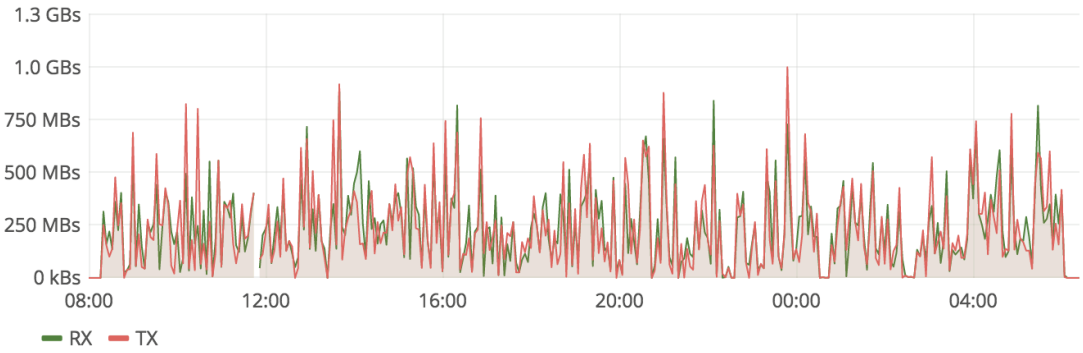


Figure 5.19 Concurrent executions of use case A: Network RX/TX

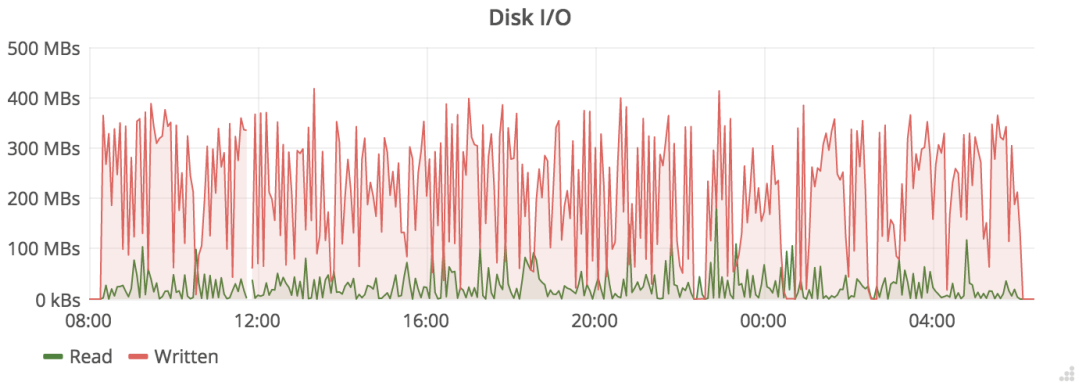


Figure 5.20 Concurrent executions of use case A: Disk I/O

Number of compute nodes

Another aspect of scalability that we evaluated was how our system would react to various numbers of compute nodes and how the workflows from our two use cases would perform. The specific scenario given in Table 5.4 describes the test we performed.

Source	Cloud infrastructure, administrators
Stimulus	New compute nodes are made available to the system
Environment	Normal operation
Artefacts	Whole system
Response	The system makes use of the capacity offered by the added compute nodes
Response measure	The time needed to execute a workflow decreases with the number of nodes added

Table 5.4 Scalability Specific Scenario B - Number of compute nodes

We applied this scenario to both use cases. We executed the workflows on 6, 9, 12, 15 and 18 compute nodes. On each node we repeated the test five times to calculate mean and median. Table 5.5 shows the results for use case A. The median values for each node are plotted in Figure 5.21 in a semi-logarithmic scale.

	6 nodes	9 nodes	12 nodes	15 nodes	18 nodes
Run #1	4h 54m	3h 19m	2h 33m	2h 10m	1h 53m
Run #2	4h 55m	3h 19m	2h 32m	2h 11m	1h 51m
Run #3	4h 55m	3h 19m	2h 34m	2h 09m	1h 53m
Run #4	4h 57m	3h 19m	2h 32m	2h 10m	1h 54m
Run #5	4h 56m	3h 20m	2h 32m	2h 09m	1h 53m
Mean	*4h 55m*	*3h 19m*	*2h 33m*	*2h 10m*	*1h 53m*
Median	*4h 55m*	*3h 19m*	*2h 32m*	*2h 10m*	*1h 53m*

Table 5.5 Execution times of use case A on a varying number of compute nodes

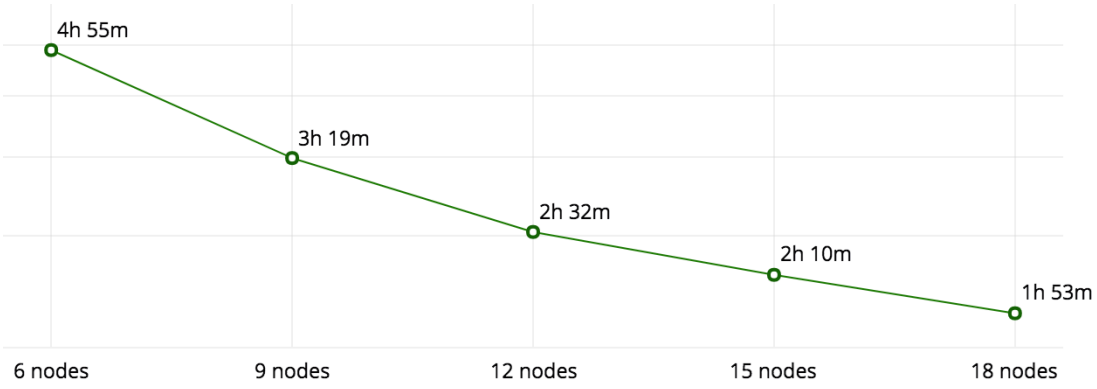


Figure 5.21 Median of the execution times of use case A on a varying number of compute nodes (semi-logarithmic)

We can state that the workflow scales almost linearly. For example, the run on 12 nodes takes almost half as long as the one on 6 nodes. The graph shows, however, a slight curve indicating that there is going to be a point where the I/O overhead will be too high and the performance will not improve even if more compute nodes are added. Looking at the DSL script for this workflow again, the reason for this behaviour becomes apparent. There is only one 'for' expression. Since all iterations are independent from each other, they can theoretically be executed in parallel. The only limitation seems to be the number of compute nodes that can process tasks in parallel at a certain point in time. In theory, the point where performance could not be improved any more by merely adding compute nodes would be at 529, which is the number of input point clouds. In practise, overhead caused by I/O and scheduling in the JobManager leads to a slight degradation.

We performed a similar test for use case B. Table 5.6 shows the collected timings and Figure 5.22 the plotted median values. Use case B performs slightly worse than use case A in terms of scalability. The figure shows a clear curve progression. There is actually no difference between the execution times on 15 and 18 nodes, which indicates that 15 nodes are the optimum and performance cannot be improved any more beyond this point. This behaviour is firstly due to the overhead of I/O operations and scheduling. The workflow reads almost all data at the beginning (in the first 'for' expression) and works with a large number of very small files later on. Secondly, the behaviour is also to a large extent caused by the fact that the two merge operations are single-threaded and cannot be parallelised. Even if more compute nodes are added, the merge operations will take the same time. The only possibility to improve performance would be to scale vertically which means adding better hardware components (i.e. faster CPUs and hard drives).

	6 nodes	9 nodes	12 nodes	15 nodes	18 nodes
Run #1	47m 55s	41m 52s	37m 48s	35m 31s	36m 29s
Run #2	48m 28s	40m 32s	37m 27s	35m 36s	35m 49s
Run #3	48m 00s	40m 27s	37m 07s	35m 11s	35m 26s
Run #4	47m 39s	40m 15s	37m 30s	35m 16s	35m 05s
Run #5	47m 40s	40m 44s	37m 01s	36m 25s	35m 02s
Mean	*47m 56s*	*40m 46s*	*37m 22s*	*35m 36s*	*35m 34s*
Median	*47m 55s*	*40m 32s*	*37m 27s*	*35m 31s*	*35m 26s*

Table 5.6 Execution times of use case B on a varying number of compute nodes

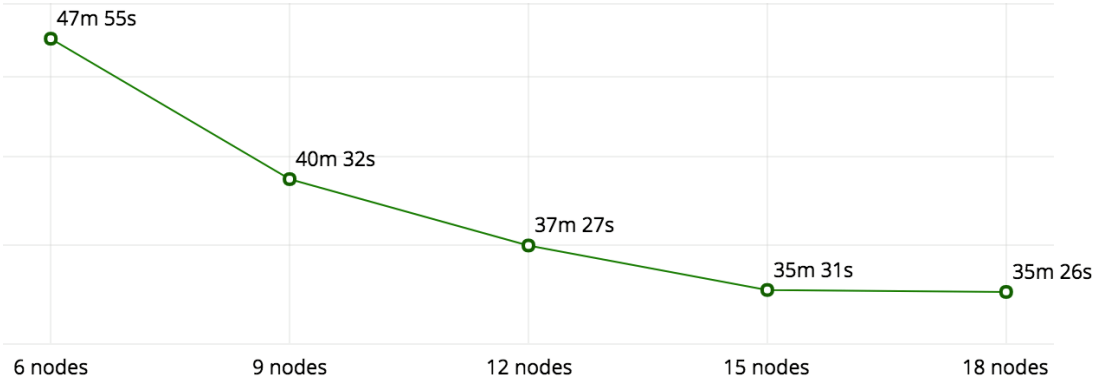


Figure 5.22 Median of the execution times of use case B on a varying number of compute nodes (semi-logarithmic)

Data volume

The third response measure for our scalability evaluation is data volume. Table 5.7 describes a specific scenario for this.

Source	Users, GIS experts, data providers
Stimulus	Execution of workflows to process arbitrarily large data volumes
Environment	Normal operation
Artefacts	Whole system
Response	The system successfully executes the workflows
Response measure	The system does not become overloaded and continues to operate normally. The system's behaviour is the same regardless of how much data it has to process.

Table 5.7 Scalability Specific Scenario C - Data volume

Above, we have already shown that our system is able to process arbitrary sizes of data. When we tested concurrent workflow executions earlier, we processed the whole data set of use case A twelve times. This makes a total of 120.63 × 12 = 1,41 TiB of data. We have shown that the

system behaves similar to a single workflow run. This is due to the fact that our system does not interact with data directly. In order to find limitations, one would have to evaluate single processing services and determine their scalability. We have done such an evaluation within the IQmulus research project (see Kießlich, Krämer, Michel, Holweg, & Gierlinger, 2016).

5.3.3 Availability

In order to evaluate whether our system can continue to operate in case of faults occurring during a workflow run, we defined the specific scenario given in Table 5.8.

Source	Network
Stimulus	Random refused connections and timeouts
Environment	Unstable operation
Artefacts	Whole system
Response	The system successfully executes the workflow from use case B
Response measure	1) The system should still be able to finish the workflow execution, even if there is a fault 2) The workflow execution might take longer as usual but should produce the same results.

Table 5.8 Availability Specific Scenario

We executed the workflow from our use case B on 18 compute nodes and simulated a very unreliable network by randomly blocking connections to compute nodes with the Linux kernel firewall. We used `iptables` to configure the firewall. For example, the following command adds a rule to block outgoing TCP connections to a node with the IP address `192.168.0.26`:

```
iptables -A OUTPUT -p tcp -d 192.168.0.26 -j DROP
```

Figure 5.23 shows the cumulative number of errors from which the JobManager was able to recover—i.e. it was able to detect an error and to reschedule work to another available compute node. The chart shows errors that occurred during the submission of process chains to the Processing Connector and during status polling.

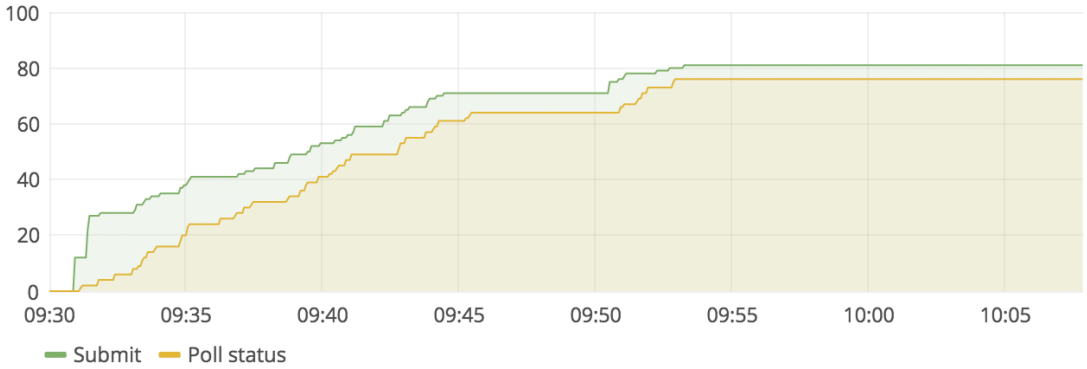


Figure 5.23 Use case B: Recoverable errors

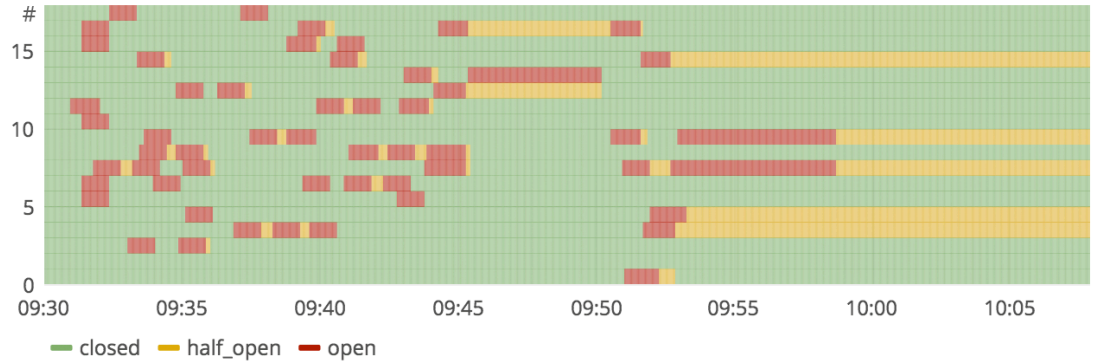


Figure 5.24 Use case B: Circuit breaker states per node

As described in Section 3.8, we implemented the Circuit Breaker pattern to avoid unnecessary calls to compute nodes that are currently unavailable. Figure 5.24 shows the states of the circuit breakers per compute node. The figure is a matrix with 18 rows (one row per node). Each of these rows represents a circuit breaker state over time. The green colour means the circuit breaker for the respective node was closed (connections were permitted). Red means the circuit breaker was open (connections were not allowed), and yellow means the circuit breaker was in half open state.

We configured each circuit breaker to close when its node could not be reached two or more times. After 60 seconds the circuit breaker should return to the half open state. From there it should either immediately return to the open state if the node was still unavailable, or change to the closed state if the first connection attempt was successful.

The whole workflow run took *41 minutes and 46 seconds*. This is about 6 minutes slower than the average run time from Section 5.3.2. However, the workflow run was successful and produced the same results as with a more reliable network connection. The JobManager was able to reschedule process chains if the compute nodes they were assigned to became unavailable.

Note that in Figure 5.24 five circuit breakers stayed in the half open state until the end of the workflow and beyond. This is due to the fact that circuit breakers only return to the closed state if there was at least one successful connection attempt. Since there was only one process chain to be executed at the end of the workflow (the second merge operation) no further connection attempts were made and the circuit breakers stayed half open until more work was assigned to their nodes in subsequent workflow runs.

5.3.4 Modifiability

For the modifiability quality attribute we identified three requirements in Section 2.3.2:

1. The users want to control how our system processes data (i.e. they want to create their own custom workflows)

2. The members of the system development team want to add new features (i.e. modify the system)

3. Developers of geospatial algorithms want to integrate their processing services into our system

Creating custom workflows

In order to allow users to create custom workflows for geospatial processing, we created a Domain-Specific Language and implemented a workflow editor in Chapter 4, *Workflow Modelling*. Table 5.9 describes a specific scenario related to this.

Source	Users
Stimulus	Create a custom workflow to process data
Environment	Runtime
Artefacts	Workflow editor
Response	The custom workflow is stored in the system and is available for execution
Response measure	It should be possible to create custom workflows for different use cases and execute them

Table 5.9 Modifiability Specific Scenario A - Creating custom workflows

This specific scenario can be realised with our system. We created workflows for use cases A and B in Section 5.2.1 and 5.2.2 respectively. We presented the workflow scripts in our Domain-Specific Language and executed the workflows to demonstrate that our system is actually able to translate them to runnable process chains. A qualitative discussion on our Domain-Specific Language is given below in Section 5.4.

Modifying the system

In order to facilitate modifiability, we designed our system based on the microservice architectural style. Microservices are loosely coupled and run as separate processes. This should allow system developers to modify and redeploy a single service without having to interrupt the operation of the rest of the system. We used the specific scenario in Table 5.10 to evaluate this quality attribute.

Source	System developers, integrators
Stimulus	Add, remove or modify functionality to the system. Change technologies, modify configurations, etc.
Environment	Compile time, build time, runtime
Artefacts	Code, interfaces, configurations
Response	The modification is made and deployed
Response measure	It should be possible to make modifications to the system without having to rebuild and redeploy it as a whole

Table 5.10 Modifiability Specific Scenario B - Modifying the system

As described in Section 1.8, we applied our approach in practise within the IQmulus research project. Software development in this project was very agile. We deployed an initial prototype of our system to the Cloud early on, so that users could test it and provide feedback. The prototype was continuously improved. In order to allow users to regularly test the system, we kept it running until the end of the project (and further) without any notable downtime.

In busy phases of the project we deployed new features and improvements several times per day. The microservice architecture was very beneficial for us because it allowed us to modify one or more services and to deploy new versions during runtime without the users even noticing it.

The only issue we had was that, initially, we could not update the JobManager while a workflow was running. We had to deploy the component redundantly and introduce mechanisms for fault-tolerance as well as the possibility to resume workflows after a restart. One of the keys to this was the fact that the JobManager is stateless and that information about running workflows and process chains are kept persistently in a database.

On the other hand, due to the fact that our microservices ran in isolated processes, we could modify any service but the JobManager at any time and redeploy it even if a workflow was currently running. We were even able to replace processing services used in a workflow while it was being executed. This was a great benefit for us that would not have been possible with a monolithic system. For further information on the evaluation of the automatic deployment of our system components see Section 5.3.6.

Integrating new processing services

One of the core requirements from the processing service developers we worked with was that it should be possible to integrate new services into the system, without having to modify the actual code of the system or to interrupt its operation (see Table 5.11).

Source	Processing service developers
Stimulus	Add, remove or modify a processing service
Environment	Runtime
Artefacts	Processing services
Response	The new version of the processing service is deployed
Response measure	It should be possible to add, remove or modify processing services without having to rebuild and redeploy the whole system

Table 5.11 Modifiability Specific Scenario C - Integrating new processing services

As described above, in the IQmulus research project we deployed a prototype of our system early on and gradually added geospatial processing services. At the end of the project we had integrated a total of 88 processing services which had been developed by partners distributed over various European countries (see Sections 5.3.5 and 5.3.6). Many of these services were improved and updated over the time and therefore deployed many times during normal operation of our system. As mentioned above, we were able to redeploy fixed versions of erroneous processing services used in a workflow while it was running. All the service developers had to do for this was to upload their service together with updated metadata to our artefact repository (see Section 2.11.2). The JobManager automatically picked new service versions up from the repository and deployed them to the Cloud.

5.3.5 Development distributability

One of the core requirements specified in Chapter 2, *Architecture* was that it should be possible that distributed teams work on different components of our architecture and integrate them at

a central location to a running system. The quality attribute that describes this requirement is *development distributability*—the possibility to distribute software development to independent teams. We validated that our system implementation has this quality attribute within the IQmulus research project. The specific scenario for development distributability (Table 5.12) reflects this.

Source	Developers working in the IQmulus project
Stimulus	Develop system components in distributed teams
Environment	–
Artefacts	Processing services, core system services
Response	Core system service and processing services act together and form our system (in this case the IQmulus platform)
Response measure	Independent teams from 12 institutions distributed over 7 European countries can develop software components and integrate them into our system on their own

Table 5.12 Development Distributability Specific Scenario

As stated earlier, in the IQmulus project we integrated a total of 88 processing services. In addition, we developed 7 core system services and deployed at least 9 external services. We had 12 different institutions from 7 European countries working on the project.

The software development was structured into teams. We worked according to the "You build it, you run it" principle (O'Hanlon, 2006) which means that the teams had a defined set of services for which they were responsible which not only included software development and bug fixing, but also operational aspects such as the integration into our system and the deployment. They also had direct contact to the end-users in our project and were involved in the requirements analysis as well as in support. This allowed them to build services that met the requirements of the people who applied them later in their workflows. In addition, the teams could react quicker to changing customer requirements or reported bugs and deploy updated versions of their services independently and at any time.

The teams were structured into groups working together on workflows from three domains: land, urban and marine. At the end of the project we realised nine different use cases with our system. Most of the processing services were only used in a specific use case, but some of them—the more generic ones—were also used multiple times across workflows.

The fact that teams worked independently and that there was no single person who integrated all services into the system required high discipline and caused communication effort that should not be underestimated. For example, we had to have bi-weekly or sometimes weekly telephone conferences, as well as separate online meetings and regular in-person meetings to coordinate the component integration. However, we believe that the efforts were still less than in a monolithic system and that we were able to use the microservice architectural style to our advantage. In a monolithic system there are often dependencies between components that grow stronger over time until they cannot be removed any more. Two dependent components then essentially become one, and the monolith becomes harder to maintain. With our microservice architecture we were able to keep flexibility and maintainability as well as independence between teams from the beginning of the software development until the end of the project (and beyond).

5.3.6 Deployability

Similar to the development distributability quality attribute, the specific scenario for the evaluation of the deployability of our system is defined within the context of the IQmulus research project (Table 5.13).

Source	System developers and processing service developers working in the IQmulus project
Stimulus	Deploy the whole system, update single services, or change configuration
Environment	Initial deployment, normal operation
Artefacts	Whole system, 104 individual services, configuration
Response	The system is fully operational
Response measure	1) The deployment process is fully automated 2) All 104 services are up and running 3) The modified configuration is in effect

Table 5.13 Deployability Specific Scenario

As mentioned earlier, in the IQmulus project we had a total of 88 processing services. In addition, we developed 7 core system services: the main user interface, the data access service, the workflow editor, the workflow service, the catalogue service (consisting of data catalogue and service catalogue), the JobManager, as well as the processing connector. We also deployed 9 external services. This includes services for monitoring and logging such as Prometheus, Grafana, Logstash, Elasticsearch, Kibana, as well as the distributed file system GlusterFS, Hadoop, Spark, and our artefact repository Artifactory.

In summary, we had 104 different components. Many of them had to be deployed multiple times to different virtual machines in the Cloud (such as the processing connector that runs on every compute node). In addition, the individual services required a lot of configuration. This included system configuration that we had to modify or service-specific configuration files.

As described in Section 2.11.3, we used the IT automation tool Ansible (Red Hat, 2017) to keep the effort of deploying and maintaining different versions of software components and configurations at a minimum. With this tool, we were able to re-deploy the whole system including all core services, as well as the external ones, with a single command. As described in Section 2.11.2, the 88 processing services were stored in our artefact repository and deployed automatically by the JobManager.

Since tasks in Ansible are idempotent, we could re-run the deployment at any time and keep the whole infrastructure in a consistent state. We kept the infrastructure description and the configurations in a code repository and under version control in order to always be able to trace back changes and to revert them if necessary (see Loukides, 2012). Similarly, the artefact repository Artifactory had a version control system that allowed us to keep track of individual versions of processing services.

In Ansible automated deployment is specified in so-called *playbooks*. One playbook describes a single component. In total, we had 42 playbooks for the system core services, the external services and the configurations. Artifactory hosted 2,207 artefacts. Most of them were different versions of processing services.

The fact that our services were microservices running in separate processes also allowed us to deploy individual services separately. This applied to the core system services and the external

services, which we managed with Ansible, but also to the processing services we kept in Artifactory and which were deployed automatically by the JobManager.

5.3.7 Portability

The final quality attribute we evaluated was portability. According to the requirements defined in Chapter 2, *Architecture*, it should be possible to deploy our system to various platforms. We tried to avoid vendor lock-ins and used technologies such as Java and Docker to isolate individual software components and to make them platform-independent. Table 5.14 describes a specific scenario to evaluate the portability of our system.

Source	Business professionals, customers, IT operations
Stimulus	The system should be deployed to a certain environment
Environment	Initial deployment
Artefacts	Whole system
Response	The system is fully operational
Response measure	The system can be deployed to two environments: an OpenStack Cloud and a VMware cluster

Table 5.14 Portability Specific Scenario

We used the automated deployment approach described in the previous Section 5.3.6 to implement this specific scenario. First, we deployed the system to the OpenStack Cloud described in Section 5.1. In addition, during the IQmulus project we deployed three instances of our system to a VMware cluster hosted by the Fraunhofer Institute for Computer Graphics Research IGD, Darmstadt, Germany. One instance was for testing and development, the second for production use within the project, and the third was a reduced system that could be made available to the public for demonstration. In all cases, the deployment worked without problems and required only little changes to the Ansible playbooks.

Since our system does not have any specific requirements, it could in fact be transferred to other environments too. The target platform only has to have some kind of support for virtual machines on which we can install our services with Ansible. The individual components of our system are also platform-independent. Most of the core services run in the Java Virtual Machine. The processing services have been containerised with Docker and packed into images that bring their own operating system, file system, dependencies and configuration.

5.4 Stakeholder requirements

In the following we review the stakeholder requirements defined in Section 2.3.1. We summarise the demands from the people (or roles) who have an interest in our system and discuss to which degree these demands are satisfied.

5.4.1 Users (GIS experts)

GIS experts need to work with large geospatial data sets, but their local workstations often lack processing power and storage capabilities. Our system gives them access to the Cloud and therefore virtually unlimited storage space and computational power. It also allows them to share their data sets with colleagues from other departments or even with other institutions or authorities.

GIS experts are used to desktop GIS solutions that offer a number of spatial operations and processing algorithms. Our microservice architecture allows a wide range of processing services to be integrated. These services can cover the functionality of a desktop GIS. Our system can therefore be considered a Cloud-based GIS.

With the workflow management and editing capabilities of our system, the GIS experts can specify how their data should be processed. In typical desktop GIS solutions they have to deal with general-purpose programming languages. The Domain-Specific Language we presented in this work allows them to focus on the workflow definition without requiring deep knowledge of programming. Our system hides technical details about the infrastructure the defined workflows are executed on and transparently generates a strategy to make best use of available resources.

The Domain-Specific Language we created in Chapter 4, *Workflow Modelling* covers our use cases from Section 1.8 but also other scenarios. In the IQmulus research project we used the same DSL to implement workflows for nine different use cases from the land, urban or marine domain. The language itself only contains a few generic keywords but can be extended by registering additional processing services. The human-readable names from the service metadata become part of the language. New services can directly be used via the generic 'apply' keyword. If more high-level language constructs are required, the language grammar has to be extended.

We designed the language to be easy to use and to hide technical details related to the infrastructure or the fact that services are potentially run in parallel in a distributed system. For example, our language only contains immutable variables (constants). This avoids many pitfalls of concurrent programming such as shared write access to resources. The limited expressiveness makes it easier for users to learn and understand our language, as well as to avoid technical details of distributed computing. However, it can—at least at first glance—also limit the system's use for very specific cases. For example, in the IQmulus research project we had a case where a user was struggling with the definition of a suitable workflow. The user was used to general-purpose languages and missed features such as arbitrary loops or mutable variables. After analysing the requirements thoroughly, we were able to implement the workflow with our language but the initial learning curve was very high for this user.

Nevertheless, our language has proven to be very efficient in practise and we could cover many use cases. Even though this thesis is about the processing of geospatial data, only a few elements of our language are related to this domain. In fact, we believe, due to its modularity based on service metadata, the same language could be used for use cases from other domains as well. In any case, the method for DSL modelling we presented in Chapter 4, *Workflow Modelling* can be used to create a new language or to modify ours and to adapt it to specific requirements.

5.4.2 Users (Data providers)

Data providers have similar overall requirements as GIS experts but typically need to deal with different types of data sets. They also often acquire large amounts of data that they quickly need to pre-process and deliver to their customers. The Cloud allows these users to scale out and to add new computational resources if required. We have shown that our system is scalable and that it

makes best use of additional resources. With our urban modelling use case, we have also shown that—with enough computational power—we can process data faster than it was acquired.

Compared to other end-users, data providers often require a different set of processing algorithms. Our system is very modular and the functionality of an instance of our system depends on the deployed processing services. Our system can be installed in various configurations targeting different user groups. This allows us to cover requirements from many use cases. Although it is optimised for geospatial data processing, our system could in general also be used outside the geospatial domain.

Data providers often need to perform the same tasks multiple times. For example, regularly acquired LiDAR data sets need to be processed always in the same way. Our system allows data providers to define workflows once and later re-use them many times. In this respect, our Domain-Specific Language helps them create workflows, but more importantly it allows them to understand existing workflows and to decide whether they are suitable or not. This is particularly important if they need to deal with a large number of pre-defined workflows from which they select a specific one that fits a certain data set or use case.

One drawback of the approach to process geospatial data in the Cloud is that data is given away to an external infrastructure provider that needs to be considered honest but curious (Krämer & Frese, 2019). Geospatial data may contain confidential information (e.g. about public infrastructure or about citizens) that should not be stored or processed in a public Cloud without additional security measures preventing unauthorised access. Data providers need to be sure that their data, which is the foundation of their business, cannot be stolen by third parties. The fact that storing data in the Cloud is potentially insecure is, however, not a drawback of our work, but of the concept of the Cloud in general. As mentioned earlier, a comprehensive security concept addressing this issue is beyond the scope of this thesis. We refer to our work on secure Cloud-based storage for geospatial data, which we conducted in parallel with this thesis (Hiemenz & Krämer, 2018).

5.4.3 Members of the system development team

The members of the system development team aim for creating a system that is maintainable and extensible. In Section 5.3.4 we have shown that new functionality can be easily added to our system and that the isolation between our microservices can help find and fix bugs. The microservice architectural style even allows individual components to be redeployed while the system is running. This enables continuous integration and hence gives end-users direct access to new functionality and other improvements.

A microservice architecture can, however, become very complex to maintain. Compared to a monolithic application, the sheer number of individual components that are deployed to various distributed nodes in the Cloud can make it hard to keep an overview. As described in Section 2.1.2, dividing the services into bounded contexts can help tackling the complexity. Regarding operations, in our system we use monitoring and distributed logging to observe the state of our services and the infrastructure. We also use IT automation for deployment and to maintain an overview of the services and the nodes they run on. We apply the *Infrastructure as Code (IaC)* approach (see Section 2.11.3) to be able to trace back changes to our system and its configuration.

5.4.4 Developers of spatial processing algorithms

Developers of spatial processing algorithms (or processing services) are often people with different backgrounds (such as mathematics, geography, or physics) who have little knowledge of distrib-

uted programming. Our architecture provides these developers with guidelines how to develop processing services that can be safely executed concurrently in a distributed environment (see Section 2.6.1).

Our architecture allows a wide range of processing services to be integrated and to be executed in parallel. This even applies to single-threaded algorithms that have not been developed for the Cloud. With our architecture developers are able to reuse existing services they created earlier and into which they have already put a lot of knowledge and effort. These services can be integrated without fundamental modifications.

As shown in Section 5.3.5, the microservice architectural style allows distributed teams of developers to create individual software components (including processing services) and to integrate them at a central location. This enables many parties to contribute their knowledge to our system and to reasonably extend its functionality.

Coordinating such a distributed development effort can, however, also be very complex. It requires regular communication so that all contributing teams are aware of architectural specifications such as the guidelines for processing services described in Section 2.6.1. It also requires that these specifications are actually followed. For example, the fact that processing services need to be idempotent is a key concept of our architecture. If the results generated by the services are not reproducible, our mechanisms for fault tolerance will not work reliably.

Furthermore, the distributed development approach requires high discipline with regards to the quality of software artefacts and at which point they can be considered releasable or deployable. According to the "You build it, you run it" principle (O'Hanlon, 2006) contributing teams therefore become responsible for the whole lifecycle of their software artefacts including development, testing, deployment and support. Despite the additional complexity and efforts, this way of development opens new possibilities. Independent teams can be a lot more agile and release new versions of their components when they think they are ready, without needing to comply with an artificial release schedule that applies to the whole distributed application. By providing support for their components, the developers get into direct contact with the end-users and can therefore create software that better satisfies their requirements. This helps reduce uncertainty and improves the overall project result. Finally, the microservice architectural style encourages teams to use different technologies that suit their specific skills and requirements best, instead of needing to rely on a technology stack that applies to the whole distributed application. This can reduce the learning curve developers have when they first join a project.

5.4.5 Integrators

Integrators are responsible for connecting core system components and processing services to a running application. To this end, they require the software artefacts to be stored in a central repository and to follow a unified version number scheme. They also require well-defined interfaces or at least machine-readable interface descriptions such as the metadata for our processing services.

As described above, within the IQmulus research project we integrated more than a hundred microservices. We used the software artefact repository Artifactory to store binaries of all system components and processing services in multiple versions. We used the semantic versioning scheme to be able to tell if an update of a software artefact was compatible to the previous version or not. This helped us reduce the integration effort since we could avoid thorough compatibility tests when we needed to update individual components. However, it also required high discipline from the developers to assign correct version numbers. Although the semantic versioning scheme has become common practise in industry in recent years, many of the processing service developers we worked with had no background in computer science and did not know this scheme. In these

cases, it was important to communicate why such a scheme is required and what could potentially go wrong if they did not comply with it.

The possibility to integrate arbitrary processing services by describing their interface in a simple JSON file (i.e. service metadata) was beneficial for us. We were able to reuse existing algorithms and services without requiring fundamental modifications. This allowed to us to extend the overall functionality of the system quickly without having to develop basic processing algorithms from scratch. Compared to a monolithic application where everything is already integrated at the moment the code is uploaded to the central source repository, integrating loosely coupled services can however be very complex. As mentioned in Section 2.3.1, all developers should be directly involved in integrating their components in order to better understand the specific requirements of a distributed environment.

Another notable aspect that helps integrate and operate a large number of services is containerisation. As described earlier, our processing services run in Docker containers, which isolates them from other software running on the same virtual machine. It also allows processing services to depend on different Linux versions or libraries, without getting into conflict with each other.

5.4.6 Testers

As described in Section 2.3.1, the microservice architectural style allows individual parts of the system (i.e. the services) to be tested separately. This can help identify issues early on before the services are integrated into the system. However, a microservice architecture also requires tests at other levels (e.g. integration tests or end-to-end tests). Even if there are thorough tests for a single service, it does not mean the service will behave as expected when it is integrated into the system and needs to communicate with other services. In addition, in order to be sure that the distributed application satisfies the needs of the end-users, UI tests or acceptance tests are required.

Compared to a monolithic application, testing can become very complex and time-consuming in a microservice architecture. It is important to find the right balance between test coverage and the level of confidence one can achieve with testing. As described in Section 2.3.3, a thorough concept for testing a microservice architecture such as ours is beyond the scope of this work. For more information on this topic we refer to Newman (2015, Chapter 7). Nevertheless, from our experience from the IQmulus research project, we can state that testing requires high discipline in a microservice architecture and many developers prefer to focus on implementing features than on making sure each and every corner case is covered. One of the benefits of the microservice architectural style is that new functionality has a short time to market, which means it can be implemented and provided to customers quickly. In order to ensure the stability of the integrated system is acceptable without slowing down the delivery process, many companies go a different path and do not put too much effort into testing. Instead, they implement a highly automated deployment pipeline that allows them to quickly revert service updates if a problem occurs in production using strategies such as *Canary Releases* or *Blue-Green Deployments* (Humble & Farley, 2010).

5.4.7 IT operations

The IT operations group is responsible for deploying an integrated system into production and to ensure continuous operation. To this end, they need automated deployment processes as well as means to monitor the system and the infrastructure.

As described above, we have used the IT automation tool Ansible to automatically deploy our system to production. This has worked very well and allowed us to make updates up to several times per day in a short time. The microservice architectural style additionally enabled us to deploy individual parts of our system separately without having to restart the whole system (*Zero-Downtime Releases*). Key to this is that everything needs to be automated, no configurations on the virtual machines are changed manually, and no software artefacts are installed without using the IT automation tool. This requires discipline from the IT operations group and the developers and can be enforced by disallowing manual SSH access to the virtual machines.

In order to monitor the system and the infrastructure, we used the tools Prometheus and Grafana. In this chapter we presented several figures that have been created with these tools. Monitoring was important for us in two ways. It helped us during development to optimise the system so that it makes best use of available Cloud resources. In addition, it is key to a continuous and smooth operation. Grafana allows for creating a dashboard where all important metrics are directly available. Through configurable alerts one can be notified about problems immediately in order to be able to react in a short time. The same applies to distributed logging. As described above we deployed Logstash, Elasticsearch and Kibana to be able to analyse log files of distributed services at a central place and to get immediate notifications about possible issues.

5.4.8 Business professionals

In Section 2.3.1 we differentiated between two types of people who have a business interest in our system: system resellers and managers of GIS projects. Both have similar requirements regarding the quality of the results our system produces, the time it takes to generate the results, and the amount of human interaction involved. They are also concerned about costs, in particular regarding maintenance and operations.

We have shown that our system is capable of running pre-defined workflows in an automated way without requiring human interaction, other than selecting the data sets to process and setting initial parameters for the processing services. We have also shown that our system is scalable and that additional computational resources can decrease the time it takes to process large data sets. The quality of the results depends on the individual processing services. Due to the microservice architecture, they can be easily replaced by improved algorithms in the future, if the current quality should not suffice.

Regarding maintenance and operations, we have shown that due to the high degree of automation we employ, the effort to update the system and to ensure smooth operation is kept at a reasonable level. The microservice architecture also allows new functionality to be added and updates to be deployed without affecting the rest of the system. Business professionals require a short time to market, which our system can offer as described above.

5.5 Objectives of the thesis

In this section we evaluate our solution against the objectives we defined at the beginning of our thesis in Section 1.4. The aim of this is to validate that our system is able to satisfy the general requirements from the two user groups from our problem statement in Section 1.3. In order to do so, we first relate the stakeholder requirements to our quality attributes (see Table 5.15).

	Performance	Scalability	Availability	Modifiability	Development distributability	Deployability	Portability
Users (GIS experts and data providers)	●	●	⊘	⊘			
System development team				●	●	⊘	
Developers of spatial processing algorithms	●	●		●	●	⊘	
Integrators				●	●	⊘	⊘
Testers				⊘	⊘	●	
IT operations			●	●		●	⊘
Business professionals	●	●	●	⊘	⊘	●	●

Table 5.15 A matrix which shows the relation between stakeholders and quality attributes

The table is directly derived from the stakeholder requirements defined in Section 2.3.1. It shows to what extent the individual stakeholders require a specific quality attribute of our system. The circle filled with black ● means the quality attribute is a primary concern, and the white circle ⊘ indicates a secondary concern.

From this table we can deduce that all quality attributes of our architecture map to stakeholders. The evaluation results from the previous sections show that in our system implementation all quality attributes are fulfilled and that the stakeholder requirements are satisfied.

We now map stakeholders to the two user groups from our problem statement in Section 1.3, namely GIS users and developers/researchers creating spatial processing algorithms (see Table 5.16).

	GIS users	Developers/researchers
Users (GIS experts and data providers)	●	
System development team		●
Developers of spatial processing algorithms		●
Integrators		●
Testers	●	
IT operations		⊘
Business professionals	⊘	⊘

Table 5.16 A matrix which maps stakeholders to user groups from our problem statement

The black circle ✔ means the stakeholder role can be directly mapped to one of the user groups. The white circle ⊘ indicates that a stakeholder can only be partly mapped to a user group or mapped to both groups.

These tables show that the two user groups defined in the objectives of our thesis can be mapped to stakeholders, and that all stakeholder requirements are satisfied by our system. *In consequence, we can now deduce that the requirements from our two user groups are satisfied and that the objectives of our thesis have been reached.* We created a system that has a user interface to process large geospatial data in the Cloud without requiring users to have expertise in distributed computing. The system is extensible so that it can cover the same functionality as a desktop GIS. It also offers the possibility to execute workflows. Existing processing algorithms can be integrated through a generic interface. The system can deploy, orchestrate and parallelise the services without requiring the developers to have expertise in Cloud Computing or to redesign their algorithms.

5.6 Summary

In this chapter we have presented the results from evaluating our architecture and its implementation. We have performed a quantitative evaluation where we first defined specific scenarios for each quality attribute our system should have, and then validated if the system satisfies the criteria given in the scenarios. We then revisited the stakeholder requirements formulated in Chapter 2, *Architecture* and discussed in which way our system meets them. We also evaluated whether the main objectives of the thesis and the general requirements from the two user groups from our problem statement are met.

Our quantitative evaluation was based on two use cases representing real-world problems from the areas of urban planning and land monitoring. We were able to show that our architecture is suitable to execute workflows from these use cases. A notable result is that the goal of the urban planning use case could be reached. We were indeed able to process large 3D point clouds faster than they were acquired. This allows municipalities and mapping authorities to efficiently utilise the data collected by laser mobile mapping systems (LMMS) and to perform tasks such as environmental monitoring based on up-to-date information. Regarding the land monitoring use case, we were able to process a large geospatial data set in the Cloud within a short amount of time. As described in Section 1.8, users from the Liguria Region reported that it took them several days to process the data on their workstations, but with our system, as shown in Section 5.2.2, it only takes about 35 minutes. This opens new possibilities for the users as it allows them to perform data analysis faster and to better prepare against environmental catastrophes.

In summary, the results of our evaluation were positive. We were able to show that our system meets all requirements in terms of quality attributes and stakeholder needs. In the next chapter we follow up on this and present conclusions and future research perspectives for our work.

6 Conclusions

With the availability of high-precision devices for mobile mapping and airborne laser scanning as well as the growing number of satellites collecting high-resolution imagery, the amount of geospatial data is becoming increasingly large. In addition, the number of consumer devices equipped with GPS sensors has rapidly grown in recent years and is expected to do so in the future. The same applies to stationary as well as mobile IoT devices.

This large amount of geospatial data needs to be processed in order to be useful for real-world applications. However, spatial algorithms are inherently complex and often require a long time to run. The data volume and the processing capabilities needed to process it exceed the capacities of workstations typically used in administrations, agencies, institutions and companies acting in the geospatial domain. Geospatial data has been recognised as Big Data, which imposes major problems for users of desktop Geographic Information Systems (GIS).

There is a paradigm shift in the geospatial community towards the Cloud which offers virtually unlimited space and computational power and is at the same time flexible, resilient, and inexpensive. However, the Cloud has not reached broad acceptance within the community yet. One of the major problems is that Cloud-based GIS products do not cover the functionality of their counterparts on the desktop. In addition, Cloud Computing is, in many aspects, still too complicated for GIS users, as well as developers and researchers providing spatial processing algorithms. The latter, in particular, are faced with new challenges regarding distributed computing but do not have the knowledge yet to tackle them.

6.1 Research results

In this thesis we have presented a software architecture for the processing of large geospatial data which helps both GIS users and algorithm developers leverage the possibilities of Cloud Computing. The architecture is based on microservices. It is modular and allows developers to integrate existing processing services, even if they were not specifically designed to be executed in the Cloud. Efficient application development can be carried out in a distributed manner by developers and researchers who work independently and contribute their services to a centralised system. Our architecture is able to automatically deploy these services to compute nodes and to orchestrate them to distributed processing workflows, so that algorithm developers without an IT background

do not need to learn concepts of distributed computing but can focus on the algorithmics. The processing workflows can be defined with an editor that is based on a Domain-Specific Language (DSL). This language is easy to learn for domain users and hides the technical details of distributed computing. This allows users to harness the capabilities of the Cloud and to focus on *what* should be done instead of *how*. These benefits allow for an enhanced experience within the group of stakeholders, which includes users, developers, analysts, and managers.

In addition to the above, we have presented the results from a thorough quantitative and qualitative evaluation, in which we validated our architecture against stakeholder requirements and quality attributes. We were able to show that our approach satisfies all requirements. According to our research design described in Section 1.7 we identified a problem, defined objectives for our research, presented a design artefact that serves as a solution for the problem, demonstrated and evaluated its utility and quality, and communicated our results in the scientific community.

In summary, we can conclude that the results from our work provide evidence to support our research hypothesis:

> A microservice architecture and Domain-Specific Languages can be used to orchestrate existing geospatial processing algorithms, and to compose and execute geospatial workflows in a Cloud environment for efficient application development and enhanced stakeholder experience.

6.2 Contributions

The contributions of this thesis can be classified into three pillars. We have created a *software architecture* contributing to the geospatial community and market by providing a means to process large data sets. In addition, we presented a workflow-based approach to the *processing of large geospatial data* that contributes to the areas of workflow management systems and service orchestration. Finally, we described an approach to *workflow modelling* based on a Domain-Specific Language and contributed our novel method for DSL modelling.

6.2.1 Architecture

Our software design is based on the microservice architectural style. Compared to monolithic applications, a microservice architecture is flexible and maintainable. In addition, it offers advantages over the Service-Oriented Architecture (SOA), in particular in terms of service deployment and execution. While services in an SOA may run in a single application container, microservices are isolated programs running in their own processes and serving a specified purpose. The high degree of isolation as well as the loose coupling of microservices helps create a distributed system in the following ways:

- **Scalability.** Our architecture has proven to be scalable in various dimensions. We were able to successfully integrate a high number of services and to deploy multiple instances of them to distributed compute nodes in the Cloud. Our architecture also allowed us to process large amounts of data and to increase performance by scaling out horizontally.

- **Modifiability.** Our system can be adapted to various use cases through the Domain-Specific Language and the rule-based system for workflow management. Loose coupling of microservices enables sustainable software development. We have shown that processing services from independent developers can be integrated to extend the functionality of our system.

- **Development distributability.** The microservice architectural style enabled distributed development. In the IQmulus research project we were able to implement a system based on our architecture that comprised more than a hundred services from twelve partners located in seven different European countries. Creating a stable system with such a range of functionality would have been impossible with a centralised, monolithic application.

- **Availability.** A microservice architecture allows for an easier implementation of various stability and availability patterns. We deployed multiple instances of our services redundantly to avoid Single Points of Failure (SPOF). In addition, we employed strategies such as the Circuit Breaker pattern to isolate services and to avoid cascading failures. The high degree of automation we used when deploying services, enabled us to keep our system constantly operating while we further developed it.

Maintaining a large distributed system can be challenging. For example, our architecture requires a certain degree of discipline from the processing service developers. We defined guidelines for service integration, but failing to follow them can undermine some of our concepts. Aspects such as fault tolerance or parallelisation can be affected if processing services do not work reproducibly and are not isolated. Additional discipline is required because our decentralised approach to create software requires service developers to not only be responsible for development, but also for deployment, operations, and user support. On the upside, this opens up new possibilities as developers can work more flexibly and independently without having to follow a strict release plan, for example. In addition, the loose coupling of microservices allows the developers to use technologies they are familiar with instead of being required to stick to a centralised technology stack.

High discipline is also required from the core system developers. A large microservice architecture can become very complex to handle. Distributed deployment of a large number of services, as is the case here, requires a high degree of automation. In addition, solutions for distributed logging and monitoring are mandatory in order to maintain an overview of the system, the deployed service instances and their current state. We only recommend a microservice architecture for systems that are large enough to justify the additional maintenance effort and that would be impossible to implement as a monolith. In our case, as stated above, the microservice architectural style was important to satisfy our requirements, and its benefits offset the challenges.

6.2.2 Data processing

The second pillar of this thesis was the workflow-based data processing. GIS users often try to automate recurring tasks by creating scripts in general-purpose programming languages with their desktop solutions. However, as mentioned above, the large amount of geospatial data exceeds the capacities of current workstations and the Cloud offers virtually unlimited storage space and computational power, but a system that is able to execute workflows for the processing of large geospatial data in the Cloud does not exist yet. More generic solutions for Big Data processing support creating single algorithms but not complex workflows.

- **Service integration.** We have worked with developers who had no background in computer science but were successfully able to integrate existing services into our architecture without having to learn distributed programming. Instead, they could focus on their algorithms. Our approach to service integration and orchestration is based on a lightweight interface description (service metadata). It is generic and does not require developers to implement a specific interface. Instead, they can describe how their services are called and integrate them as-is.

- **Service orchestration.** We have successfully implemented our approach to distributed workflow management. Our architecture is able to orchestrate geospatial processing services, to deploy them to Cloud nodes and to parallelise their execution, even if they were not specifically made for a distributed environment.

- **Dynamic workflow management.** We contributed to the state of the art with our approach to dynamic workflow execution without a priori design-time knowledge. Existing workflow management systems require all variables to be known before the workflow execution starts. However, geospatial applications often need a more dynamic solution. Very large data sets need to be split into smaller tiles that can then be processed in parallel on multiple compute nodes. At the end, the results need to be merged together. This pattern is visible in our use case B (land monitoring) where the input LiDAR strips are first partitioned along drainage boundaries and then processed independently. Later they are stitched together into a single 3D surface. The partitioning process can generate an indefinite number of tiles. The exact number will only be known during the workflow execution. With our architecture, we were able to successfully execute this kind of dynamic workflow.

- **Rule-based workflow execution.** Our approach to manage workflows is based on production rules. This makes our architecture configurable and adaptable to various use cases. For example, we can create rules that prevent users from accessing data sets or processing services if they do not have an appropriate license. The rule system can also be used to orchestrate services and to create executable process chains. Compared to other methods to describe service interfaces, our lightweight approach with generic service metadata does not define strict semantics for input and output ports. Our rule system can detect if the output of one service is not compatible to the input of a subsequent one (due to different file formats, different spatial reference systems, etc.) and add appropriate conversion services. In addition, our rule system can generate hints for our scheduler to leverage data locality and to reduce network traffic.

6.2.3 Workflow modelling

As mentioned in the previous section, desktop GIS products offer the possibility to automate recurring tasks with scripts written in a general-purpose programming language. GIS users need to understand this language and to have some experience in programming to make full use of it. If an automated task (or a workflow) should be executed in the Cloud, new questions arise, in particular regarding the selection of the right algorithms, the distribution of data, and the problem modelling.

- **DSL for workflow modelling.** With our approach to workflow modelling based on a Domain-Specific Language (DSL), GIS users with no background in computer science or distributed computing are enabled to automate tasks and process data in the Cloud.

- **Novel DSL modelling method.** In order to create such a language, we have used a novel approach to DSL modelling. This approach makes use of best practises from software engineering in order to derive the vocabulary for the language. It is incremental and iterative. Individual steps happen in collaboration with domain users in order to get direct feedback.

The DSL we created in this thesis covers two use cases from the urban and land domains. However, the basic vocabulary is generic and can be applied to other use cases too. The domain-specific terms are modular and can be replaced or extended depending on the actual requirements. Since human-readable names in our lightweight service metadata automatically become part of the DSL,

the processing services that are integrated into an instance of our architecture define its appearance. This means if we would remove all processing services from our system that are specific to the geospatial domain and add services from another domain instead, we could use our architecture for completely new applications. If more specific expressions are required in the DSL we can apply our modelling method to this domain.

6.3 Future Work

Given the potential of Cloud-based solutions for geospatial applications, we see many possibilities to continue our research and to improve our architecture. These possibilities cover aspects such as scalability and automatic data partitioning, as well as applying our solution to broader use cases. One aspect for further research was already mentioned in Section 2.7.4 where we discussed the possibility to replace our distributed file system (DFS) by a structured and indexed storage solution for geospatial data called GeoRocket. The DFS has proven to be flexible and a good way to transfer files from one processing service to another. One problem that remains, however, is related to *data partitioning*. In our architecture we rely on the fact that the input data sets are already partitioned into smaller tiles and that these can be processed independently. This enables us to distribute the tiles to multiple compute nodes and to parallelise the processing. Problems arise if the individual tiles are too large to be handled by one node or if a certain algorithm requires multiple tiles to generate correct results—e.g. a feature extraction algorithm detecting shorelines that span across several tiles. In our use case dealing with land monitoring, the input data set is first repartitioned and indexed before it can be triangulated. A more intelligent tiling approach could be implemented with GeoRocket which is a data store that automatically splits imported data into smaller chunks. The indexing and querying features of GeoRocket enable a much more precise access to data. In future work we will investigate the use of this solution for our architecture and whether we can implement additional use cases with it. Since GeoRocket is a data store similar to an object storage solution such as AWS S3 with an HTTP interface, data access will be slower than with our distributed file system. We will investigate the performance impact and evaluate whether it is outweighed by the benefits.

Another area where we can improve the scalability of our architecture and contribute to the current state of the art in the area of workflow management systems is our *internal data model for workflows*. At the moment, we create a directed acyclic graph (DAG) to determine the dependencies between workflow tasks. This is an approach followed by many workflow management systems. The problem is that such a graph can become very large as soon as the number of files to be processed, and hence the number of tasks, grows. In our case, this problem is aggravated by the fact that we support dynamic workflows whose DAGs can change while they are executed. Such large graphs can grow beyond the size of the memory available to our JobManager. In addition, it can take a long time to traverse them. Existing workflow management systems try to solve this problem by offering the possibility to split a workflow into sub-workflows that have smaller DAGs and are executed subsequently. However, this is a workaround and the workflow management systems often leave it up to the user to decide whether to use it or not, which introduces additional complexity. We have already started working on a different approach that creates the graph incrementally based on the number of available compute nodes and memory (Hellhake, 2017). We will continue working on this approach as we see the potential to achieve high scalability with a low memory footprint.

Our approach to workflow management is based on a rule system. We were able to achieve a high flexibility and configurability with this but also observed some difficulties. With a high number of rules and many facts in the working memory, it can become hard to maintain an overview and it is easy for a developer to make mistakes and create rules that drastically affect

the performance of the process chain generation—e.g. if a rule contains conditions that require the rule system to create a cross product of facts. In this work we were able to show that rules are suitable for the selection of data, processing services and nodes. In future work, we will try to focus on this and *reduce the complexity of our rule system in order to avoid performance pitfalls*. Many rules can be easily transferred to imperative code, which will improve the overall scalability and performance of our system without sacrificing configurability.

Finally, we will *investigate the use of our architecture for other applications in the geospatial domain*, in particular those where streams of data need to be processed continuously. For example, for land monitoring it might be suitable to regularly process satellite imagery that is produced every other week and to incorporate results from analysing daily weather data. Since our architecture is based on events, we can react immediately to newly imported data and start the processing in order to always be able to provide up-to-date results to the user. This approach can be combined with the data storage based on GeoRocket, which is also event-based and has a concept of a secondary data store containing processing results. We think such an approach provides a lot of potential for applications dealing with geospatial information, not only point clouds (such as in this thesis) but also raster images and spatio-temporal data.

6.4 Final remarks

Before we started with research on the topic of processing large geospatial data about five years ago, the Cloud did not have the same market acceptance as today. This particularly applies to the geospatial market where the Cloud is still a niche solution and has only just begun to gain acceptance. The same is true for the microservice architectural style. Although microservices have gained a high momentum recently in the IT industry, they were relatively unknown five years ago. As we designed our architecture we needed an approach to create a large, Cloud-based system that consisted of many isolated services that are developed independently and in a distributed manner. The microservice architectural style became popular at almost the same time and, as such, helped us design our architecture.

Similarly, we see great potential for the Cloud in the geospatial market. In recent years we have observed a growing number of research projects dealing with Cloud-based geospatial applications. The same applies to the industry that offers more and more solutions in this area. In this thesis we have focussed on two groups, GIS users and developers of geospatial solutions. Although the Cloud has not reached broad acceptance in the geospatial market, we can observe that the users are increasingly becoming the market drivers. In particular, this is with respect to their demand to process large geospatial data that exceeds the capabilities of their workstations and that developers have started to meet this with innovative Cloud-based solutions.

The paradigm shift from desktop GIS to the Cloud is already taking place but there is still a long way to go. We believe that this thesis documents a major step, but more work needs to be done, not only on a conceptual or technical level, but in particular, in increasing the acceptance of the Cloud amongst users and developers.

Appendix A. Combined DSL grammar

The following listing shows the combined PEG (Parsing Expression Grammar) for the Domain-Specific Language created in Chapter 4, *Workflow Modelling* for use cases A and B. The grammar can be compiled using the open-source parser generator PEG.js (Majda, 2016).

```
start
  = SP* (statements SP*)?

statements
  = statement ( SP+ statement )*

statement
  = with / for / operation

with
  = WITH SP+ dataset SP+ block

for
  = FOR SP+ EACH SP+ dataset SP+ DO SP+ statements (SP+ yield)?
    SP+ END ( SP+ AS SP+ NAME )?

yield
  = YIELD ( SP+ ref )?

operation
  = special_operation / param_operation

special_operation
  = (VISUALIZE / STORE) (SP+ ref)?

param_operation
  = operation_name (SP+ NAME)? operation_with? operation_using?
    operation_as?

operation_name
  = APPLY / CREATE / EXCLUDE / EXTRACT / REMOVE / REORDER / RESAMPLE
  / SPLIT / UPDATE

operation_with
  = SP+ WITH SP+ dataset ( SP+ AND SP+ dataset )*

operation_using
  = SP+ USING SP+ params

operation_as
  = SP+ AS SP+ NAME

block
  = DO SP+ statements SP+ END
```

```
dataset
  = placeholder / (RECENT SP+)? ref

params
  = param ( SP+ AND SP+ param )*

param
  = NAME SP* ":" SP* expression

expression
  = NUMBER / string

placeholder
  = "[" NAME "]"

ref
  = objectRef / varRef

objectRef
  = NAME SP* "." SP* ref

varRef
  = NAME

string
  = '"' STRING_CHAR* '"'

KEYWORD
  = AND / APPLY / AS / CREATE / DO / EACH / END / EXCLUDE / EXTRACT
  / FOR / RECENT / REMOVE / REORDER / RESAMPLE / SPLIT / STORE
  / UPDATE / USING / VISUALIZE / WITH / YIELD

NAME           = !KEYWORD [_a-zA-Z] NAME_MORE*
NAME_MORE      = [_a-zA-Z0-9]
NUMBER         = [0-9]+ ( "." [0-9]+ )?
STRING_CHAR    = !["\\\r\n] . / "\\" ESCAPE_CHAR
ESCAPE_CHAR    = ["\\bfnrtv]
COMMENT        = ("#" / "//") (!"\n" .)*
SP             = [ \t\n\r] / COMMENT

AND            = "and"            !NAME_MORE
APPLY          = "apply"          !NAME_MORE
AS             = "as"             !NAME_MORE
CREATE         = "create"         !NAME_MORE
DO             = "do"             !NAME_MORE
EACH           = "each"           !NAME_MORE
END            = "end"            !NAME_MORE
EXCLUDE        = "exclude"        !NAME_MORE
EXTRACT        = "extract"        !NAME_MORE
FOR            = "for"            !NAME_MORE
RECENT         = "recent"         !NAME_MORE
REMOVE         = "remove"         !NAME_MORE
REORDER        = "reorder"        !NAME_MORE
RESAMPLE       = "resample"       !NAME_MORE
SPLIT          = "split"          !NAME_MORE
STORE          = "store"          !NAME_MORE
UPDATE         = "update"         !NAME_MORE
USING          = "using"          !NAME_MORE
VISUALIZE      = "visualize"      !NAME_MORE
               / "visualise"      !NAME_MORE
WITH           = "with"           !NAME_MORE
YIELD          = "yield"          !NAME_MORE
```

Appendix B. Scientific work

This appendix provides an overview of scientific work I conducted in the past. This includes a list of peer-reviewed publications, extended abstracts, and posters authored or co-authored by me, as well as relevant talks given by me, relevant work in scientific projects, and awards I received for my research.

B.1 Journal papers

Hiemenz, B., & Krämer, M. (2018). Dynamic Searchable Symmetric Encryption in Geospatial Cloud Storage. *International Journal of Information Security*. Submitted, under review.

Krämer, M., & Frese, S. (2019). Implementing Secure Applications in Smart City Clouds Using Microservices. Submitted, under review.

Krämer, M., & Senner, I. (2015). A Modular Software Architecture for Processing of Big Geospatial Data in the Cloud. *Computers & Graphics*, *49*, 69–81. https://doi.org/10.1016/j.cag.2015.02.005

B.2 Conference proceedings

Böhm, J., Bredif, M., Gierlinger, T., Krämer, M., Lindenbergh, R., Liu, K., … Sirmacek, B. (2016). The IQmulus Urban Showcase: Automatic Tree Classification and Identification in Huge Mobile Mapping Point Clouds. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLI-B3*, 301–307. https://doi.org/10.5194/isprs-archives-XLI-B3-301-2016

Coors, V., & Krämer, M. (2011). Integrating Quality Management into a 3D Geospatial Server. In *Proceedings of the 28th Urban Data Management Symposium UDMS* (pp. 7–12). ISPRS.

Dambruch, J., & Krämer, M. (2014). Leveraging Public Participation in Urban Planning with 3D Web Technology. In *Proceedings of the Nineteenth International ACM Conference on 3D Web Technologies (Web3D)* (pp. 117–124). ACM. https://doi.org/10.1145/2628588.2628591

Krämer, M. (2014). Controlling the Processing of Smart City Data in the Cloud with Domain-Specific Languages. In *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC)* (pp. 824–829). IEEE.

Krämer, M., Dummer, M., Ruppert, T., & Kohlhammer, J. (2011). Tackling Uncertainty in Combined Visualizations of Underground Information and 3D City Models. In *GeoViz Hamburg 2011 Workshop*. HafenCity University, Hamburg and International Cartographic Association (ICA): Commission on GeoVisualization.

Krämer, M., & Gutbell, R. (2015). A case study on 3D geospatial applications in the Web using state-of-the-art WebGL frameworks. In *Proceedings of the Twentieth International ACM Conference on 3D Web Technologies (Web3D)* (pp. 189–197). ACM. https://doi.org/10.1145/2775292.2775303

Krämer, M., Haist, J., & Reitz, T. (2007). Methods for Spatial Data Quality of 3D City Models. In *Proceedings of the 5th Italian Chapter Conference*. European Association for Computer Graphics (Eurographics).

Krämer, M., & Kehlenbach, A. (2013). Interactive, GPU-Based Urban Growth Simulation for Agile Urban Policy Modelling. In *Proceedings of the 27th European Conference on Modelling and Simulation (ECMS)* (pp. 75–81). European Council for Modelling and Simulation.

Krämer, M., Ludlow, D., & Khan, Z. (2013). Domain-Specific Languages for Agile Urban Policy Modelling. In *Proceedings of the 27th European Conference on Modelling and Simulation (ECMS)* (pp. 673–680). European Council for Modelling and Simulation.

Krämer, M., & Stein, A. (2014). Automated Urban Management Processes: Integrating a Graphical Editor for Modular Domain-Specific Languages into a 3D GIS. In *Proceedings of the 19th International Conference on Urban Planning, Regional Development and Information Society REAL CORP* (pp. 99–108). Schwechat, Austria: CORP – Competence Center of Urban and Regional Planning.

Malewski, C., Dambruch, J., & Krämer, M. (2015). Towards Interactive Geodata Analysis through a Combination of Domain-Specific Languages and 3D Geo Applications in a Web Portal Environment. In *Proceedings of the 20th International Conference on Urban Planning, Regional Development and Information Society REAL CORP* (pp. 609–616). Schwechat, Austria: CORP – Competence Center of Urban and Regional Planning.

Reitz, T., Krämer, M., & Thum, S. (2009). A Processing Pipeline for X3D Earth-based Spatial Data View Services. In *Proceedings of the 14th International ACM Conference on 3D Web Technologies (Web3D)* (pp. 137–145). ACM.

Ruppert, T., Dambruch, J., Krämer, M., Balke, T., Gavanelli, M., Bragaglia, S., … Kohlhammer, J. (2015). Visual Decision Support for Policy Making – Advancing Policy Analysis with Visualization. In C. G. Reddick (Ed.), *Policy practice and digital science: Integrating complex systems, social simulation and public administration in policy research* (pp. 321–353). Springer. https://doi.org/10.1007/978-3-319-12784-2_15

Thum, S., & Krämer, M. (2011). Reducing Maintenance Complexity of User-centric Web Portrayal Services. In *Proceedings of the Sixteenth International ACM Conference on 3D Web Technologies (Web3D)* (pp. 165–172). ACM.

B.3 Extended abstracts and posters

Dummer, M., Krämer, M., Ruppert, T., & Kohlhammer, J. (2011). Visualizing Uncertain Underground Information for Urban Management. In *Working with Uncertainty Workshop, IEEE VisWeek 2011*.

Krämer, M., & Klien, E. (2010). Visualisation and integration of 3D underground information into city models with DeepCity3D / Visualisierung und Integration von dreidimensionalen Untergrunddaten und Stadtmodellen mit DeepCity3D. In *GeoDarmstadt* (pp. 327–328).

Quak, E., Spagnuolo, M., Holweg, D., Brédif, M., Krämer, M., Nguyen Thai, B., … Kießlich, N. (2015). IQmulus Scalability Testing - First Results. *Workshop on GeoBigData, ISPRS Geospatial Week*.

B.4 Relevant project deliverables

Krämer, M., & Senner, I. (2015). *IQmulus public project deliverable D2.4.2 - Processing DSL Specification - final version*.

Krämer, M., Skytt, V., Patane, G., Kießlich, N., Spagnuolo, M., & Michel, F. (2015). *IQmulus public project deliverable D2.3.2 - Architecture design - final version*.

Krämer, M., Zulkowski, M., Plabst, S., & Kießlich, N. (2014). *IQmulus public project deliverable D3.2 - Control Components - vertical prototype release*.

B.5 Relevant talks

This section includes a selection of relevant talks from recent years.

IQmulus Infrastructure (2016). Virtual Geoscience Conference VGC, Bergen, Norway.

IQmulus Infrastructure (2016). Geospatial, Mathematical and Linked Big Data, IQmulus workshop at European Data Forum 2016, Eindhoven, Netherlands.

IQmulus - Automatische Photogrammetrische Prozesse in der Cloud (2016). Münchener GI-Runde, Runder Tisch GIS, Munich, Germany.

Smart City Clouds (2015-2016). InGeoForum workshop series on Cloud for geospatial applications.

Visualizing Large 3D city models in the Web (2015). 19th International ACM Conference on 3D Web Technologies Web3D, Vancouver, B.C., Canada.

B.6 Relevant work in scientific projects

Scientific Manager in the IQmulus research project (2012-2015). 7th Framework Programme of the European Commission, call identifier FP7-ICT-2011-8, under the grant agreement no. 318787.

As the Scientific Manager my role within the IQmulus project included defining the software architecture, leading the software development and monitoring the scientific progress. In this respect I had to coordinate more than twelve distributed teams of researchers and software developers from various partners spread over several European countries. In addition, I was responsible for the development of the core system components including a workflow management component as well as a workflow editor based on a Domain-Specific Language.

Scientific Manager in the urbanAPI research project (2011-2014). 7th Framework Programme of the European Commission, call identifier FP7-ICT-2011-7, under the grant agreement no: 288577.

Similar to the IQmulus project, I held the role of the Scientific Manager and was as such responsible for designing the overall software architecture of the project and coordinating the development. I was also responsible for creating a Domain-Specific Language and an editor for the modelling of urban policy rules.

B.7 Awards

Best paper award at *Computer Graphik Abend 2016* in the category *Impact on Business* for Krämer, M., & Senner, I. (2015) A Modular Software Architecture for Processing of Big Geospatial Data in the Cloud.

Best paper award at *Computer Graphik Abend 2015* in the category *Impact on Society* for Dambruch, J., & Krämer, M. (2014) Leveraging Public Participation in Urban Planning with 3D Web Technology.

Best paper award at the *19th International ACM Conference on 3D Web Technologies (Web3D)* for Dambruch, J., & Krämer, M. (2014) Leveraging Public Participation in Urban Planning with 3D Web Technology.

Honourable mention at *Computer Graphik Abend 2012* for Coors, V., & Krämer, M. (2011) Integrating Quality Management into a 3D Geospatial Server.

Appendix C. Teaching

This appendix contains a list of courses given by me as a lecturer at the Technische Hochschule Mittelhessen in Gießen, as well as student theses I supervised over the last years.

C.1 Courses

Funktionale Programmierung – Functional Programming (2011, summer term). Bachelor's programme. Department for Mathematics, Natural sciences and Information Technology (MNI). TH Mittelhessen, Gießen.

Verteilte Systeme – Distributed Systems (2012, summer term). Master's programme. Department for Mathematics, Natural sciences and Information Technology (MNI). TH Mittelhessen, Gießen.

Cloud Computing und Big Data – Cloud Computing and Big Data (2013, summer term). Masters's programme. Department for Mathematics, Natural sciences and Information Technology (MNI). TH Mittelhessen, Gießen.

Verteilte Systeme – Distributed Systems (2014, summer term). Master's programme. Department for Mathematics, Natural sciences and Information Technology (MNI). TH Mittelhessen, Gießen.

C.2 Supervising activities

Baas, T. (2011). *Integration einer Qualitätssicherungskomponente in ein 3D-Geoinformationssystem*. Stuttgart, Hochschule für Technik, Bachelor's Thesis.

Frese, S. (2015). *Secure Cloud-Based Risk Assessment for Urban Areas: Sichere cloudbasierte Risikoanalyse für Stadtgebiete*. Darmstadt, TU, Master's Thesis.

Hellhake, T. (2017). *Building a scalable and fault-tolerant cloud architecture for the distributed execution of workflows*. Gießen, TH Mittelhessen, Master's Thesis.

Hiemenz, B. (2016). *Authentication and Searchable Symmetric Encryption for Cloud-based Storage of Geospatial Data*. Darmstadt, TU, Master's Thesis.

Min, Q. (2009). *3D Visualization of Zoning Maps Using CityServer3D and Generative Modeling Language*. Stuttgart, Hochschule für Technik, Master's Thesis.

Pompetzki, R. (2010). *Integration und Visualisierung von Vegetationsdaten der Landeshauptstadt Mainz in ein 3D-Geoinformationssystem*. Wiesbaden, Hochschule RheinMain, Diplomarbeit.

Reuter, D. (2010). *Aufbau einer Produktlinie für ein Geoinformationssystem und Bereitstellung unterstützender Werkzeuge*. Giessen-Friedberg, FH, Bachelor's Thesis.

Reuter, D. (2012). *Optimierung von Verarbeitungsprozessen und Erstellung einer Cloud-Architektur für ein Geoinformationssystem*. Campus Friedberg, TH Mittelhessen, Master's Thesis.

Sajenko, A. (2017). *Seminatürliche Abfragesprache für Geodaten*. Gießen, TH Mittelhessen, Master's Thesis.

Schäfer, M. (2016). *Konzeption und Realisierung einer vertikalen Microservice Architektur für einen Online-Geodatenkatalog*. Gießen, TH Mittelhessen, Master's Thesis.

Senner, I. (2015). *Rule-based Process Orchestration: An Expert System for the Dynamic and Infrastructure-Independent Generation of Geospatial Processing Chains*. Gießen, TH Mittelhessen, Master's Thesis.

Stein, A. (2010). *Stadtplanung und -marketing anhand interaktiver digitaler Bebauungspläne*. Giessen-Friedberg, FH, Bachelor's Thesis.

Stein, A. (2012). *Anforderungsanalyse und Entwicklung eines Editors für grafische domänenspezifische Sprachen*. Campus Friedberg, TH Mittelhessen, Master's Thesis.

Thum, S. (2009). *Szenarioorientierte Darstellung heterogener Geodaten*. Gießen-Friedberg, FH, Master's Thesis.

Zimmermann, R. (2013). *Digitale Wasserzeichen für die webbasierte Präsentation von 3D-Stadtmodellen*. Gießen, TH Mittelhessen, Master's Thesis.

Bibliography

Agarwal, D., & Prasad, S. K. (2012). Lessons Learnt from the Development of GIS Application on Azure Cloud Platform. In *Proceedings of the 5th IEEE International Conference on Cloud Computing (CLOUD)* (pp. 352–359). https://doi.org/10.1109/CLOUD.2012.140

Ahmed, R., Lee, A., Witkowski, A., Das, D., Su, H., Zait, M., & Cruanes, T. (2006). Cost-based Query Transformation in Oracle. In *Proceedings of the 32nd International Conference on Very Large Data Bases* (pp. 1026–1036). Seoul, Korea: VLDB Endowment.

Ajiy, A., Sun, X., Vo, H., Liu, Q., Lee, R., Zhang, X., … Wang, F. (2013). Demonstration of Hadoop-GIS: A Spatial Data Warehousing System Over MapReduce. In *21st ACM International Conference on Advances in Geographic Information Systems (SIGSPATIAL GIS 2013)*.

Alpers, S., Becker, C., Oberweis, A., & Schuster, T. (2015). Microservice Based Tool Support for Business Process Modelling. In *19th IEEE International Enterprise Distributed Object Computing Workshop* (pp. 71–78). https://doi.org/10.1109/EDOCW.2015.32

Alshuqayran, N., Ali, N., & Evans, R. (2016). A Systematic Mapping Study in Microservice Architecture. In *9th IEEE International Conference on Service-Oriented Computing and Applications SOCA* (pp. 44–51). https://doi.org/10.1109/SOCA.2016.15

Amaral, M., Polo, J., Carrera, D., Mohomed, I., Unuvar, M., & Steinder, M. (2015). Performance Evaluation of Microservices Architectures Using Containers. In *14th IEEE International Symposium on Network Computing and Applications* (pp. 27–34). https://doi.org/10.1109/NCA.2015.49

Amazon. (2016). Amazon States Language. Retrieved 8 March 2017, from https://states-language.net/spec.html

Amstutz, P., Crusoe, M. R., Tijanić, N., Chapman, B., Chilton, J., Heuer, M., … Stojanovic, L. (2005). *Common Workflow Language, v1.0*. https://doi.org/10.6084/m9.figshare.3115156.v2

Apfelbacher, R., & Rozinat, A. (2003). Fundamental Modeling Concepts (FMC) Notation Reference. Retrieved 29 April 2017, from http://www.fmc-modeling.org/download/notation_reference/FMC-Notation_Reference.pdf

Armstrong, R., Kumfert, G., McInnes, L. C., Parker, S., Allan, B., Sottile, M., … Dahlgren, T. (2006). The CCA component model for high-performance scientific computing. *Concurrency and Computation: Practice and Experience*, *18*(2), 215–229. https://doi.org/10.1002/cpe.911

Balalaie, A., Heydarnoori, A., & Jamshidi, P. (2016). Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture. *IEEE Software*, *33*(3), 42–52. https://doi.org/10.1109/MS.2016.64

Balis, B. (2014). Increasing Scientific Workflow Programming Productivity with HyperFlow. In *Proceedings of the 9th Workshop on Workflows in Support of Large-Scale Science WORKS '14* (pp. 59–69). Piscataway, NJ, USA: IEEE Press. https://doi.org/10.1109/WORKS.2014.10

Barros, A., Dumas, M., & Oaks, P. (2006). Standards for Web Service Choreography and Orchestration: Status and Perspectives. In C. J. Bussler & A. Haller (Eds.), *Business Process Management Workshops: BPM 2005 International Workshops, BPI, BPD, ENEI, BPRM, WSCOBPM, BPS. Revised Selected Papers* (pp. 61–74). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/11678564_7

Bass, L., Clements, P., & Kazman, R. (2012). *Software Architecture in Practice* (3rd ed.). Addison-Wesley Professional. https://doi.org/10.1145/2693208.2693252

Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., & Pérez, C. (2009). GCM: a grid extension to Fractal for autonomous distributed components. *Annals of Telecommunications*, *64*(1), 5–24. https://doi.org/10.1007/s12243-008-0068-8

Belényesi, M., & Kristóf, D. (2014). *IQmulus public project deliverable D1.2.3 - Revised User Requirements*.

Bini, O. (2008). Fractal Programming. Retrieved 21 December 2016, from https://olabini.com/blog/2008/06/fractal-programming/

Bondi, A. B. (2000). Characteristics of Scalability and Their Impact on Performance. In *Proceedings of the 2nd International Workshop on Software and Performance* (pp. 195–203). ACM. https://doi.org/10.1145/350391.350432

Bonér, J., Farley, D., Kuhn, R., & Thompson, M. (2014). The Reactive Manifesto v2.0. Retrieved 10 April 2017, from http://www.reactivemanifesto.org/

Booth, T., & Stumpf, S. (2013). End-User Experiences of Visual and Textual Programming Environments for Arduino. In Y. Dittrich, M. Burnett, A. Mørch, & D. Redmiles (Eds.), *Proceedings of the 4th International Symposium on End-User Development IS-EUD* (pp. 25–39). Berlin, Heidelberg: Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-38706-7_4

Breiman, L. (2001). Random Forests. *Machine Learning*, *45*(1), 5–32. https://doi.org/10.1023/A:1010933404324

Broad Institute. (2017). Workflow Description Language (WDL). Retrieved 8 March 2017, from https://software.broadinstitute.org/wdl/

Brooks, F. P., Jr. (1987). No Silver Bullet – Essence and Accidents of Software Engineering. *Computer*, *20*(4), 10–19. https://doi.org/10.1109/MC.1987.1663532

Burke, B. (2013). *RESTful Java with JAX-RS 2.0*. O'Reilly.

Böhm, J., Bredif, M., Gierlinger, T., Krämer, M., Lindenbergh, R., Liu, K., … Sirmacek, B. (2016). The IQmulus Urban Showcase: Automatic Tree Classification and Identification in Huge Mobile Mapping Point Clouds. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLI-B3*, 301–307. https://doi.org/10.5194/isprs-archives-XLI-B3-301-2016

Cahalane, C., McCarthy, T., & McElhinney, C. P. (2012). MIMIC: Mobile Mapping Point Density Calculator. In *Proceedings of the 3rd International Conference on Computing for Geospatial Research and Applications* (p. 15:1–15:9). ACM. https://doi.org/10.1145/2345316.2345335

Chafi, H., Sujeeth, A. K., Brown, K. J., Lee, H., Atreya, A. R., & Olukotun, K. (2011). A Domain-specific Approach to Heterogeneous Parallelism. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming* (pp. 35–46). San Antonio, TX, USA: ACM. https://doi.org/10.1145/1941553.1941561

Charfi, A., & Mezini, M. (2004). Hybrid Web Service Composition: Business Processes Meet Business Rules. In *Proceedings of the 2nd International Conference on Service Oriented Computing* (pp. 30–38). New York, NY, USA: ACM. https://doi.org/10.1145/1035167.1035173

Chaudhuri, S. (1998). An Overview of Query Optimization in Relational Systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (pp. 34–43). Seattle, Washington, USA: ACM. https://doi.org/10.1145/275487.275492

Ciuffoletti, A. (2015). Automated Deployment of a Microservice-based Monitoring Infrastructure. *Procedia Computer Science*, *68*, 163–172. https://doi.org/http://dx.doi.org/10.1016/j.procs.2015.09.232

Cleary, A., Kohn, S., Smith, S., & Smolinski, B. (1998). Language Interoperability Mechanisms for High-Performance Scientific Applications. In *Proceedings of the SIAM Workshop on Object-Oriented Methods for Interoperable Scientific and Engineering Computing* (pp. 30–39).

Cohn, M. (2009). *Succeeding with Agile: Software Development Using Scrum*. Addison Wesley.

Conway, M. E. (1968). How Do Committees Invent? *Datamation*, *14*(4), 28–31.

Coppock, J. T., & Rhind, D. W. (1991). The history of GIS. *Geographical Information Systems*, *1*, 21–43.

Cossu, R., Di Giulio, C., Brito, F., & Petcu, D. (2013). Cloud Computing for Earth Observation. In D. Kyriazis, A. Voulodimos, S. V. Gogouvitis, & T. Varvarigou (Eds.), *Data Intensive Storage Services for Cloud Environments* (pp. 166–191). IGI Global. https://doi.org/10.4018/978-1-4666-3934-8

Dambruch, J., & Krämer, M. (2014). Leveraging Public Participation in Urban Planning with 3D Web Technology. In *Proceedings of the Nineteenth International ACM Conference on 3D Web Technologies (Web3D)* (pp. 117–124). ACM. https://doi.org/10.1145/2628588.2628591

de Jesus, J., Walker, P., Grant, M., & Groom, S. (2012). WPS orchestration using the Taverna workbench: The eScience approach. *Computers & Geosciences*, *47*, 75–86. https://doi.org/http://doi.org/10.1016/j.cageo.2011.11.011

De Nicola, A., & Missikoff, M. (2016). A Lightweight Methodology for Rapid Ontology Engineering. *Communications of the ACM*, *59*(3), 79–86. https://doi.org/10.1145/2818359

De Nicola, A., Missikoff, M., & Navigli, R. (2009). A software engineering approach to ontology building. *Information Systems*, *34*(2), 258–275. https://doi.org/10.1016/j.is.2008.07.002

Dean, J., & Ghemawat, S. (2008). MapReduce: simplified data processing on large clusters. *Commun. ACM*, *51*(1), 107–113. https://doi.org/10.1145/1327452.1327492

Dearle, F. (2010). *Groovy for Domain-Specific Languages* (1st ed.). Packt Publishing.

Deelman, E., Gannon, D., Shields, M., & Taylor, I. (2009). Workflows and e-Science: An overview of workflow system features and capabilities. *Future Generation Computer Systems*, *25*(5), 528–540. https://doi.org/10.1016/j.future.2008.06.012

Deelman, E., Vahi, K., Juve, G., Rynge, M., Callaghan, S., Maechling, P. J., … Wenger, K. (2015). Pegasus: a Workflow Management System for Science Automation. *Future Generation Computer Systems*, *46*, 17–35. https://doi.org/10.1016/j.future.2014.10.008

Demantké, J., Mallet, C., David, N., & Vallet, B. (2011). Dimensionality Based Scale Selection in 3D LiDAR Point Clouds. In *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences* (Vol. XXXVIII-5/W12, pp. 97–102). ISPRS. https://doi.org/10.5194/isprsarchives-XXXVIII-5-W12-97-2011

Deutsch, P. (1994). The Eight Fallacies of Distributed Computing. Retrieved 23 November 2016, from https://blogs.oracle.com/jag/resource/Fallacies.html

Di Francesco, P., Malavolta, I., & Lago, P. (2017). Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption. In *IEEE International Conference on Software Architecture ICSA* (pp. 21–30). https://doi.org/10.1109/ICSA.2017.24

Docker Inc. (2017). Docker - Build, Ship, and Run Any App, Anywhere. Retrieved 29 May 2017, from https://www.docker.com/

D'Amato Avanzi, G., Galanti, Y., Giannecchini, R., & Bartelletti, C. (2015). Shallow Landslides Triggered by the 25 October 2011 Extreme Rainfall in Eastern Liguria (Italy) (pp. 515–519). Springer International Publishing. https://doi.org/10.1007/978-3-319-09057-3_85

Eclipse. (2017). Vert.x - A tool-kit for building reactive applications on the JVM. Retrieved 10 April 2017, from http://vertx.io/

Elasticsearch BV. (2017). Elasticsearch: RESTful, Distributed Search & Analytics. Retrieved 30 May 2017, from https://www.elastic.co/

Eldawy, A., & Mokbel, M. (2013). A Demonstration of SpatialHadoop: An Efficient Mapreduce Framework for Spatial Data. In *Proceedings of the 39th International Conference on Very Large Data Bases (VLDB)* (Vol. 6, pp. 1230–1233). VLDB Endowment. https://doi.org/10.14778/2536274.2536283

Eldawy, A., & Mokbel, M. (2014). Pigeon: A Spatial MapReduce Language. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*.

Esposito, C., Castiglione, A., & Choo, K.-K. R. (2016). Challenges in Delivering Software in the Cloud as Microservices. *IEEE Cloud Computing*, *3*(5), 10–14. https://doi.org/10.1109/MCC.2016.105

Evans, E. (2003). *Domain-Driven Design: Tacking Complexity In the Heart of Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Felter, W., Ferreira, A., Rajamony, R., & Rubio, J. (2015). An updated performance comparison of virtual machines and Linux containers. In *IEEE International Symposium on Performance Analysis of Systems and Software ISPASS* (pp. 171–172). https://doi.org/10.1109/ISPASS.2015.7095802

Fielding, R. T. (2010). *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.

Fielding, R. T., & Reschke, J. F. (Eds.). (2014). *Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content, RFC 7231*. https://doi.org/10.17487/RFC7231

Footen, J., & Faust, J. (2008). *The Service-Oriented Media Enterprise: SOA, BPM, and Web Services in Professional Media Systems*. Focal Press.

Fowler, M. (2010). *Domain Specific Languages* (1st ed.). Addison-Wesley Professional.

Fowler, M. (2012). TestPyramid. Retrieved 23 May 2017, from https://martinfowler.com/bliki/TestPyramid.html

Fowler, M. (2014). Microservices - a definition of this new architectural term. Retrieved 23 November 2016, from http://www.martinfowler.com/articles/microservices.html

Fraunhofer IGD. (2017). GeoRocket - High-performance data store for geospatial files. Retrieved 30 May 2017, from https://georocket.io/

Friis-Christensen, A., Lucchi, R., Lutz, M., & Ostländer, N. (2009). Service chaining architectures for applications implementing distributed geographic information processing. *International Journal of Geographical Information Science*, *23*(5), 561–580. https://doi.org/10.1080/13658810802665570

Gil, Y., Ratnakar, V., Kim, J., González-Calero, P. A., Groth, P. T., Moody, J., & Deelman, E. (2011). Wings: Intelligent Workflow-Based Design of Computational Experiments. *IEEE Intelligent Systems*, *26*(1), 62–72. https://doi.org/10.1109/MIS.2010.9

Gilmore, D. J., & Green, T. R. G. (1984). Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies*, *21*(1), 31–48. https://doi.org/http://dx.doi.org/10.1016/S0020-7373(84)80037-1

Goodchild, M. F. (2007). Citizens as sensors: the world of volunteered geography. *GeoJournal*, *69*(4), 211–221. https://doi.org/10.1007/s10708-007-9111-y

Google. (2017). AngularJS - Superheroic JavaScript MVW Framework. Retrieved 10 March 2017, from https://angularjs.org/

Gradle Inc. (2017). Gradle Build Tool. Retrieved 9 March 2017, from https://gradle.org/

Grafana Labs. (2017). Grafana - The open platform for analytics and monitoring. Retrieved 1 June 2017, from https://grafana.com/

Green, T. R., Petre, M., & Bellamy, R. (1991). Comprehensibility of visual and textual programs: A test of superlativism against the 'match-mismatch' conjecture. *ESP*, *91*(743), 121–146.

Hellhake, T. (2017). *Building a scalable and fault-tolerant cloud architecture for the distributed execution of workflows*. Gießen, TH Mittelhessen, Master's Thesis.

Herbst, N. R., Kounev, S., & Reussner, R. (2013). Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proceedings of the 10th International Conference on Autonomic Computing ICAC* (pp. 23–27). San Jose, CA: USENIX.

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design Science in Information Systems Research. *MIS Quarterly*, *28*(1), 75–105.

Hiemenz, B., & Krämer, M. (2018). Dynamic Searchable Symmetric Encryption in Geospatial Cloud Storage. *International Journal of Information Security*. Submitted, under review.

Hofer, C., & Ostermann, K. (2010). Modular Domain-specific Language Components in Scala. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering* (pp. 83–92). Eindhoven, The Netherlands: ACM. https://doi.org/10.1145/1868294.1868307

Hong, M., Riedewald, M., Koch, C., Gehrke, J., & Demers, A. (2009). Rule-based Multi-query Optimization. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (pp. 120–131). ACM. https://doi.org/10.1145/1516360.1516376

Hudak, P. (1998). Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse* (pp. 134–142). Washington, DC, USA: IEEE Computer Society. https://doi.org/10.1109/ICSR.1998.685738

Humble, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

ISO 19115-1. (2014). *Geographic information -- Metadata -- Part 1: Fundamentals*. International Organization for Standardization, Geneva, Switzerland.

ISO 19119. (2016). *Geographic information -- Services*. International Organization for Standardization, Geneva, Switzerland.

Jacob, J. C., Katz, D. S., Berriman, G. B., Good, J. C., Laity, A. C., Deelman, E., … Williams, R. (2009). Montage: a Grid Portal and Software Toolkit for Science-Grade Astronomical Image Mosaicking. *International Journal of Computational Science and Engineering*, *4*(2), 73–87. https://doi.org/10.1504/IJCSE.2009.026999

Jacobson, I., Booch, G., & Rumbaugh, J. (1999). *The Unified Software Development Process*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Jaeger, E., Altintas, I., Zhang, J., Ludäscher, B., Pennington, D., & Michener, W. (2005). A Scientific Workflow Approach to Distributed Geospatial Data Processing Using Web Services. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management* (pp. 87–90). Berkeley, CA, US: Lawrence Berkeley Laboratory.

Josuttis, N. M. (2009). *SOA in Practice: The Art of Distributed System Design*. O'Reilly.

Juric, B. (2010). The 'Scala is too Complex' Conspiracy. Retrieved 9 March 2017, from https://warpedjavaguy.wordpress.com/2010/08/02/the-scala-is-too-complex-conspiracy-1/

Keller, F., Tabeling, P., Apfelbacher, R., Groene, B., Gröne, B., Knoepfel, A., … Schmidt, O. (2002). Improving Knowledge Transfer at the Architectural Level: Concepts And Notations. In *Concepts and Notations, International Conference on Software Engineering Research and Practice, Las Vegas*.

Khan, Z., Anjum, A., & Kiani, S. L. (2013). Cloud Based Big Data Analytics for Smart Future Cities. In *Proceedings of the 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing UCC 2013* (pp. 381–386). Washington, DC, USA: IEEE Computer Society. https://doi.org/10.1109/UCC.2013.77

Khan, Z., & Kiani, S. L. (2012). A Cloud-Based Architecture for Citizen Services in Smart Cities. In *Proceedings of the 2012 IEEE/ACM 5th International Conference on Utility and Cloud Computing UCC 2012* (pp. 315–320). Washington, DC, USA: IEEE Computer Society. https://doi.org/10.1109/UCC.2012.43

Kießlich, N., Krämer, M., Michel, F., Holweg, D., & Gierlinger, T. (2016). *IQmulus public project deliverable D6.5 - Scalability Tests - Version 2*.

Kitchin, R., & McArdle, G. (2016). What makes Big Data, Big Data? Exploring the ontological characteristics of 26 datasets. *Big Data & Society*, *3*(1), 1–10. https://doi.org/10.1177/2053951716631130

Kratzke, N. (2015). About Microservices, Containers and their Underestimated Impact on Network Performance. In *Proceedings of the 6th International Conference on Cloud Computing, GRIDS and Virtualization CLOUD COMPUTING* (pp. 165–169).

Kreps, J. (2014). Questioning the Lambda Architecture. Retrieved 14 June 2017, from https://www.oreilly.com/ideas/questioning-the-lambda-architecture

Krylovskiy, A., Jahn, M., & Patti, E. (2015). Designing a Smart City Internet of Things Platform with Microservice Architecture. In *Proceedings of the 3rd International Conference on Future Internet of Things and Cloud* (pp. 25–30). https://doi.org/10.1109/FiCloud.2015.55

Krämer, M. (2014). Controlling the Processing of Smart City Data in the Cloud with Domain-Specific Languages. In *Proceedings of the 7th International Conference on Utility and Cloud Computing (UCC)* (pp. 824–829). IEEE.

Krämer, M., & Frese, S. (2019). Implementing Secure Applications in Smart City Clouds Using Microservices. Submitted, under review.

Krämer, M., Ludlow, D., & Khan, Z. (2013). Domain-Specific Languages for Agile Urban Policy Modelling. In *Proceedings of the 27th European Conference on Modelling and Simulation (ECMS)* (pp. 673–680). European Council for Modelling and Simulation.

Krämer, M., & Stein, A. (2014). Automated Urban Management Processes: Integrating a Graphical Editor for Modular Domain-Specific Languages into a 3D GIS. In *Proceedings of the 19th International Conference on Urban Planning, Regional Development and Information Society REAL CORP* (pp. 99–108). Schwechat, Austria: CORP – Competence Center of Urban and Regional Planning.

Lanig, S., Schilling, A., Stollberg, B., & Zipf, A. (2008). Towards Standards-Based Processing of Digital Elevation Models for Grid Computing through Web Processing Service (WPS). In O. Gervasi, B. Murgante, A. Laganà, D. Taniar, Y. Mun, & M. L. Gavrilova (Eds.), *Proceedings of the 2008 International Conference on Computational Science and Its Applications ICCSA* (pp. 191–203). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-69848-7_17

Lee, C., & Percivall, G. (2008). Standards-Based Computing Capabilities for Distributed Geospatial Applications. *Computer*, *41*(11), 50–57. https://doi.org/10.1109/MC.2008.468

Li, J., Humphrey, M., Agarwal, D., Jackson, K., van Ingen, C., & Ryu, Y. (2010). eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)* (pp. 1–10). https://doi.org/10.1109/IPDPS.2010.5470418

Liu, K., Boehm, J., & Alis, C. (2016). Change Detection of Mobile LiDAR Data Using Cloud Computing. *ISPRS - International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences*, *XLI-B3*, 309–313. https://doi.org/10.5194/isprs-archives-XLI-B3-309-2016

Loukides, M. (2012). *What is DevOps? - Infrastructure as Code*. O'Reilly Media.

Ludlow, D., & Khan, Z. (2012). Participatory democracy and the governance of smart cities. In *Proceedings of the 26th Annual AESOP Congress*.

Ludäscher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E., Jones, M., … Zhao, Y. (2006). Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience*, *18*(10), 1039–1065. https://doi.org/10.1002/cpe.v18:10

Ludäscher, B., Weske, M., McPhillips, T., & Bowers, S. (2009). Scientific Workflows: Business as Usual? In U. Dayal, J. Eder, J. Koehler, & H. A. Reijers (Eds.) (pp. 31–47). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-03848-8_4

Lutz, M., Lucchi, R., Friis-Christensen, A., & Ostländer, N. (2007). A Rule-Based Description Framework for the Composition of Geographic Information Services. In F. Fonseca, M. A. Rodríguez, & S. Levashkin (Eds.), *Proceedings of the 2007 Second International Conference on GeoSpatial Semantics GeoS* (pp. 114–127). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-76876-0_8

Majda, D. (2016). PEG.js - Parser Generator for JavaScript. Retrieved 10 March 2017, from https://pegjs.org/

Malawski, M. (2016). Towards Serverless Execution of Scientific Workflows - HyperFlow Case Study. In *Proceedings of the 11th Workshop on Workflows in Support of Large-Scale Science* (pp. 25–33).

Malawski, M., Meizner, J., Bubak, M., & Gepner, P. (2011). Component Approach to Computational Applications on Clouds. *Procedia Computer Science*, *4*, 432–441. https://doi.org/10.1016/j.procs.2011.04.045

Manolescu, D. (2000). Micro-Workflow: A Workflow Architecture Supporting Compositional Object-Oriented Software Development. Champaign, IL, USA: Doctoral dissertation, University of Illinois.

Marz, N., & Warren, J. (2015). *Big Data: Principles and Best Practices of Scalable Realtime Data Systems* (1st ed.). Manning Publications Co.

Mell, P. M., & Grance, T. (2011). *SP 800-145. The NIST Definition of Cloud Computing*. Gaithersburg, MD, United States: National Institute of Standards & Technology.

Mernik, M., Heering, J., & Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Computing Surveys*, *37*(4), 316–344. https://doi.org/10.1145/1118890.1118892

MongoDB Inc. (2017). The MongoDB database. Retrieved 30 May 2017, from https://www.mongodb.com/

Monnier, F., Vallet, B., & Soheilian, B. (2012). Trees Detection from Laser Point Clouds Acquired in Dense Urban Areas by a Mobile Mapping System. In *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* (Vol. I-3, pp. 245–250). ISPRS. https://doi.org/10.5194/isprsannals-I-3-245-2012

Nagios Enterprises. (2017). Nagios - The Industry Standard In IT Infrastructure Monitoringcs. Retrieved 1 June 2017, from https://www.nagios.org/

Natis, Y. V. (2003). *Service-Oriented Architecture Scenario*. Retrieved from Gartner database.

Neag, I. A., Tyler, D. F., & Kurtz, W. S. (2001). Visual programming versus textual programming in automatic testing and diagnosis. In *Proceedings of the 2001 IEEE Autotestcon. IEEE Systems Readiness Technology Conference* (pp. 658–671). https://doi.org/10.1109/AUTEST.2001.949450

Newman, S. (2015). *Building Microservices* (1st ed.). O'Reilly Media, Inc.

Nygard, M. (2007). *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf.

OASIS. (2007). Web Services Business Process Execution Language Version 2.0. Retrieved 8 March 2017, from http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

OGC. (2015). WPS 2.0 Interface Standard. Retrieved 21 April 2017, from http://docs.opengeospatial.org/is/14-065/14-065.html

Olston, C., Reed, B., Srivastava, U., Kumar, R., & Tomkins, A. (2008). Pig Latin: A Not-so-foreign Language for Data Processing. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (pp. 1099–1110). Vancouver, Canada: ACM. https://doi.org/10.1145/1376616.1376726

OMG. (2006). CORBA Component Model (CCM) - Specification v4.0. Retrieved 21 April 2017, from http://www.omg.org/spec/CCM/4.0

OMG. (2013). Business Process Model and Notation (BPMN) - Specification v2.0.2. Retrieved 8 March 2017, from http://www.omg.org/spec/BPMN/2.0.2/

OpenStack Foundation. (2017). OpenStack. Retrieved 26 June 2017, from https://www.openstack.org/

O'Hanlon, C. (2006). A Conversation with Werner Vogels. *Queue*, *4*, 14:14-14:22. https://doi.org/10.1145/1142055.1142065

Paparoditis, N., Papelard, J. P., Cannelle, B., Devaux, A., Soheilian, B., David, N., & Houzay, E. (2012). Stereopolis II: A multi-purpose and multi-sensor 3D mobile mapping system for street visualisation and 3D metrology. *Revue Française de Photogrammétrie et de Télédétection*, *200*(1), 69–79.

Patil, A. A., Oundhakar, S. A., Sheth, A. P., & Verma, K. (2004). METEOR-S Web Service Annotation Framework. In *Proceedings of the 13th International Conference on World Wide Web* (pp. 553–562). New York, NY, USA: ACM. https://doi.org/10.1145/988672.988747

Peffers, K., Tuunanen, T., Rothenberger, M., & Chatterjee, S. (2007). A Design Science Research Methodology for Information Systems Research. *Journal of Management Information Systems*, *24*(3), 45–77. https://doi.org/10.2753/MIS0742-1222240302

Peltz, C. (2003). Web services orchestration and choreography. *Computer*, *36*(10), 46–52. https://doi.org/10.1109/MC.2003.1236471

Peters-Anders, J., Loibl, W., Züger, J., Khan, Z., & Ludlow, D. (2014). Exploring population distribution and motion dynamics through mobile phone device data in selected cities - lessons learned from the UrbanAPI project. In *Proceedings of 19th International Conference on Urban Planning, Regional Development and Information Society REAL CORP* (pp. 871–876).

Petre, M. (1995). Why Looking Isn't Always Seeing: Readership Skills and Graphical Programming. *Commun. ACM*, *38*(6), 33–44. https://doi.org/10.1145/203241.203251

Pirahesh, H., Hellerstein, J. M., & Hasan, W. (1992). Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 39–48). San Diego, California, USA: ACM. https://doi.org/10.1145/130283.130294

Preston-Werner, T. (2013). Semantic Versioning 2.0.0. Retrieved 29 May 2017, from http://semver.org/spec/v2.0.0.html

Prometheus Community. (2016). Prometheus - Monitoring system & time series database. Retrieved 1 June 2017, from https://prometheus.io/

Qazi, N., Smyth, D., & McCarthy, T. (2013). Towards a GIS-Based Decision Support System on the Amazon Cloud for the Modelling of Domestic Wastewater Treatment Solutions in Wexford, Ireland. In *15th International Conference on Computer Modelling and Simulation (UKSim)* (pp. 236–240). https://doi.org/10.1109/UKSim.2013.62

Rautenbach, V., Coetzee, S., & Iwaniak, A. (2013). Orchestrating OGC web services to produce thematic maps in a spatial information infrastructure. *Computers, Environment and Urban Systems*, *37*, 107–120.

Red Hat. (2017). Ansible. Retrieved 24 May 2017, from https://www.ansible.com/

Reinsel, D., Gantz, J., & Rydning, J. (2017). *Data Age 2025 - The Evolution of Data to Life-Critical*. An IDC White Paper, Sponsored by Seagate.

Robbins, J., Krishnan, K., Allspaw, J., & Limoncelli, T. (2012). Resilience Engineering: Learning to Embrace Failure. *Communications of the ACM*, *55*(11), 40–47.

Roy, P., & Sudarshan, S. (2009). Multi-Query Optimization. In L. Liu & M. T. Özsu (Eds.), *Encyclopedia of Database Systems* (pp. 1849–1852). Boston, MA: Springer US. https://doi.org/10.1007/978-0-387-39940-9_239

Ruppert, T., Dambruch, J., Krämer, M., Balke, T., Gavanelli, M., Bragaglia, S., … Kohlhammer, J. (2015). Visual Decision Support for Policy Making – Advancing Policy Analysis with Visualization. In C. G. Reddick (Ed.), *Policy practice and digital science: Integrating complex systems, social simulation and public administration in policy research* (pp. 321–353). Springer. https://doi.org/10.1007/978-3-319-12784-2_15

Russell, N., ter Hofstede, A. H. M., Edmond, D., & van der Aalst, W. M. P. (2004a). *Workflow Data Patterns* (QUT Technical report, FIT-TR-2004-01). Brisbane: Technical report, Queensland University of Technology.

Russell, N., ter Hofstede, A. H. M., Edmond, D., & van der Aalst, W. M. P. (2004b). *Workflow Resource Patterns* (BETA Working Paper Series, WP 127). Technical report, Eindhoven University of Technology.

Russell, N., ter Hofstede, A. H. M., van der Aalst, W. M. P., & Mulyar, N. (2006). *Workflow Control-Flow Patterns: A Revised View* (BPM-06-22). Technical report, BPM Center.

Russell, N., van der Aalst, W. M. P., & ter Hofstede, A. H. M. (2016). *Workflow Patterns: The Definitive Guide*. MIT Press.

Safina, L., Mazzara, M., Montesi, F., & Rivera, V. (2016). Data-Driven Workflows for Microservices: Genericity in Jolie. In *30th IEEE International Conference on Advanced Information Networking and Applications AINA* (pp. 430–437). https://doi.org/10.1109/AINA.2016.95

Schulte, W. R. (1996). *'Service Oriented' Architectures, Part 2*. Retrieved from Gartner database.

Schulte, W. R., & Natis, Y. V. (1996). *'Service Oriented' Architectures, Part 1*. Retrieved from Gartner database.

Sheth, A. P., van der Aalst, W. M. P., & Arpinar, I. B. (1999). Processes Driving the Networked Economy. *IEEE Concurrency*, *7*(3), 18–31. https://doi.org/10.1109/4434.788776

Shewchuk, J. R. (1996). Triangle: Engineering a 2D quality mesh generator and Delaunay triangulator. In M. C. Lin & D. Manocha (Eds.), *Applied Computational Geometry Towards Geometric Engineering* (Vol. 1148 of the Lecture Notes in Computer Science, pp. 203–222). Springer Berlin Heidelberg. https://doi.org/10.1007/BFb0014497

Sirmacek, B., & Lindenbergh, R. (2015). Automatic classification of trees from laser scanning point clouds. In *ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences* (Vol. II-3/W5, pp. 137–144). ISPRS. https://doi.org/10.5194/isprsannals-II-3-W5-137-2015

Stollberg, B., & Zipf, A. (2007). OGC Web Processing Service Interface for Web Service Orchestration Aggregating Geo-processing Services in a Bomb Threat Scenario. In J. M. Ware & G. E. Taylor (Eds.), *Proceedings of the 7th International Symposium on Web and Wireless Geographical Information Systems W2GIS 2007* (pp. 239–251). Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-76925-5_18

Sujeeth, A. K., Lee, H., Brown, K. J., Rompf, T., Chafi, H., Wu, M., … Olukotun, K. (2011). OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In L. Getoor & T. Scheffer (Eds.), *ICML* (pp. 609–616). Omnipress.

Thain, D., Tannenbaum, T., & Livny, M. (2005). Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, *17*(2–4), 323–356.

Toffetti, G., Brunner, S., Blöchlinger, M., Dudouet, F., & Edmonds, A. (2015). An Architecture for Self-managing Microservices. In *Proceedings of the 1st International Workshop on Automated Incident Management in Cloud* (pp. 19–24). ACM. https://doi.org/10.1145/2747470.2747474

Topcuouglu, H., Hariri, S., & Wu, M. (2002). Performance-Effective and Low-Complexity Task Scheduling for Heterogeneous Computing. *IEEE Transactions on Parallel and Distributed Systems*, *13*(3), 260–274. https://doi.org/10.1109/71.993206

Ulrich, H. (2011). The Pros & Cons of Scala. Retrieved 9 March 2017, from http://blog.celerity.com/pros-cons-scala

van der Aalst, W. M. P., & ter Hofstede, A. H. M. (2005). YAWL: Yet Another Workflow Language. *Information Systems*, *30*(4), 245–275. https://doi.org/10.1016/j.is.2004.02.002

van der Aalst, W. M. P., ter Hofstede, A. H. M., Kiepuszewski, B., & Barros, A. P. (2003). Workflow Patterns. *Distributed and Parallel Databases*, *14*(3), 5–51. https://doi.org/10.1023/A:1022883727209

Vatsavai, R. R., Ganguly, A., Chandola, V., Stefanidis, A., Klasky, S., & Shekhar, S. (2012). Spatiotemporal Data Mining in the Era of Big Spatial Data: Algorithms and Applications. In *Proceedings of the 1st ACM SIGSPATIAL International Workshop on Analytics for Big Geospatial Data* (pp. 1–10). ACM. https://doi.org/10.1145/2447481.2447482

Vianden, M., Lichter, H., & Steffens, A. (2014). Experience on a Microservice-Based Reference Architecture for Measurement Systems. In *21st Asia-Pacific Software Engineering Conference* (Vol. 1, pp. 183–190). https://doi.org/10.1109/APSEC.2014.37

Villamizar, M., Garcés, O., Castro, H., Verano, M., Salamanca, L., Casallas, R., & Gil, S. (2015). Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *10th Computing Colombian Conference 10CCC* (pp. 583–590). https://doi.org/10.1109/ColumbianCC.2015.7333476

Villamizar, M., Garcés, O., Ochoa, L., Castro, H., Salamanca, L., Verano, M., … Lang, M. (2016). Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures. In *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing CCGrid* (pp. 179–182). https://doi.org/10.1109/CCGrid.2016.37

W3C. (2001). Web Services Description Language (WSDL) 1.1. Retrieved 8 March 2017, from https://www.w3.org/TR/2001/NOTE-wsdl-20010315

Warren, M. S., Brumby, S. P., Skillman, S. W., Kelton, T., Wohlberg, B., Mathis, M., … Johnson, M. (2015). Seeing the Earth in the Cloud: Processing one petabyte of satellite imagery in one day. In *IEEE Applied Imagery Pattern Recognition Workshop (AIPR)* (pp. 1–12). https://doi.org/10.1109/AIPR.2015.7444536

Warshaw, L. B., & Miranker, D. P. (1999). Rule-based Query Optimization, Revisited. In *Proceedings of the 8th International Conference on Information and Knowledge Management* (pp. 267–275). ACM. https://doi.org/10.1145/319950.320012

Weigand, H., van den Heuvel, W.-J., & Hiel, M. (2008). Rule-based service composition and service-oriented business rule management. In *Interdisciplinary Workshop Regulations Modelling and Deployment*.

Weinmann, M., Jutzi, B., & Mallet, C. (2014). Semantic 3D scene interpretation: A framework combining optimal neighborhood size selection with relevant features. In *ISPRS Annals of Photogrammetry, Remote Sensing and Spatial Information Sciences* (Vol. II-3, pp. 181–188). ISPRS. https://doi.org/10.5194/isprsannals-II-3-181-2014

Whitley, K. N. (1997). Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages & Computing*, *8*(1), 109–142. https://doi.org/http://dx.doi.org/10.1006/jvlc.1996.0030

Wolstencroft, K., Haines, R., Fellows, D., Williams, A., Withers, D., Owen, S., … Goble, C. (2013). The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud. *Nucleic Acids Research*, *41*(W1), W557–W561. https://doi.org/10.1093/nar/gkt328

Xin, R., Rosen, J., Zaharia, M., Franklin, M. J., Shenker, S., & Stoica, I. (2013). Shark: SQL and Rich Analytics at Scale. In *Proceedings of the ACM SIGMOD/PODS Conference*.

Yang, C., Goodchild, M. F., Huang, Q., Nebert, D., Raskin, R., Xu, Y., … Fay, D. (2011). Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing? *International Journal of Digital Earth*, *4*(4), 305–329. https://doi.org/10.1080/17538947.2011.587547

Yu, J., & Buyya, R. (2005). A Taxonomy of Workflow Management Systems for Grid Computing. *Journal of Grid Computing*, *3*(3), 171–200. https://doi.org/10.1007/s10723-005-9010-8