This is a post-peer-review, pre-copyedit version of an article published in Computers & Graphics. The final authenticated version is available online at: https://doi.org/10.1016/j.cag.2015.02.005

A modular software architecture for processing of big geospatial data in the cloud

Michel Krämer[a,b], Ivo Senner[a]

[a] *Fraunhofer Institute for Computer Graphics Research IGD, 64283 Darmstadt, Germany*
[b] *Technische Universität Darmstadt, 64283 Darmstadt, Germany*

Abstract

In this paper we propose a software architecture that allows for processing of large geospatial datasets in the cloud. Our system is modular and flexible and supports multiple algorithm design paradigms such as MapReduce, in-memory computing or agent-based programming. It contains a web-based user interface where domain experts (e.g. GIS analysts or urban planners) can define high-level processing workflows using a domain-specific language (DSL). The workflows are passed through a number of components including a parser, interpreter, and a service called job manager. These components use declarative and procedural knowledge encoded in rules to generate a processing chain specifying the execution of the workflows on a given cloud infrastructure according to the constraints defined by the user. The job manager evaluates this chain, spawns processing services in the cloud and monitors them. The services communicate with each other through a distributed file system that is scalable and fault-tolerant. Compared to previous work describing cloud infrastructures and architectures we focus on the processing of big heterogeneous geospatial data. In addition to that, we do not rely on only one specific programming model or a certain cloud infrastructure but support several ones. Combined with the possibility to control the processing through DSL-based workflows, this makes our architecture very flexible and configurable. We do not only see the cloud as a means to store and distribute large datasets but also as a way to harness the processing power of distributed computing environments for large-volume geospatial datasets. The proposed architecture design has been developed for the IQmulus research project funded by the European Commission. The paper concludes with the evaluation results from applying our solution to two example workflows from this project.

Keywords: Cloud computing, Big Data, Geoprocessing, Distributed systems, Software architectures, Domain-specific languages

1. Introduction

With the availability of modern sensors and LiDAR scanners (*Li*ght *D*etection *A*nd *R*anging) that deliver hundreds of GiB up to several TiB per hour, the world-wide volume of geospatial data grows exponentially. While processing spatial information has always been a complex and time-consuming task, this new kind of large-volume data (*Big Data* or *Big Geo Data*) requires new techniques that make use of modern technology such as multi-core programming and GPGPU programming [1]. However, such large data typically does not fit into the memory of one computer. It is often stored in a distributed manner on multiple computer systems. These computers make up the nodes of a distributed infrastructure typically referred to as a *cloud*. There is an ongoing effort to utilise the cloud for the processing of Big Geo Data and to make it available for a wide range of applications such as earth observation [2–4], environmental protection [5, 6], or urban planning for future smart cities [7–9].

An important property of clouds is their scalability which is facilitated by the fact that a cloud infrastructure offers virtually unlimited resources in terms of processing power and memory. Large data centres contain hundreds of computer systems with inexpensive, medium-sized hardware—often referred to as commodity hardware—that can work in concert to store large-volume data and to perform complex computations. Should resources become low, new nodes can be added to the cloud, typically without any system downtime. At the same time, clouds provide broad network access which makes the stored data available from virtually anywhere through thick and thin clients [10]. This is an important fact to be considered if up-to-date geospatial data is required in the field.

In this work we present a software architecture for processing of large-volume geospatial data in the cloud. Our design consists of a web-based user interface where expert users can define a processing workflow using a domain-specific language (DSL). This workflow is evaluated by an interpreter and a software component called job manager subsequently. Both components use pre-defined rules to create a processing chain that specifies execution of the workflow on a given infrastructure according to user-defined constraints. While previous work focuses on spatial data storage and provisioning [5] as well as on specific programming paradigms [3, 11, 12] we present an architecture that is modular and flexible. It supports multiple algorithm design paradigms such as MapReduce [13], actor-based programming [14] or in-memory computing [15]. Which kind of programming paradigm is used for a certain algorithm depends on the specific use case and requirements. In addition

Email address: michel.kraemer@igd.fraunhofer.de (Michel Krämer)

to that, our architecture does not rely on a specific cloud infrastructure but can be deployed to multiple ones.

The rest of this paper is organised as follows. We start with an analysis of related work and describe how our approach differentiates from other cloud architectures for geospatial processing. We also describe functional and technical requirements and how our design meets them. In the main part of this paper we present the overall architecture of our system and then describe the individual components in detail. In the following we discuss implementation challenges and present evaluation results from applying our solution to two example workflows from the IQmulus international research project funded by the European Commission. We finish the paper with a summary and conclusion.

2. Related work

While there has been a lot of work on cloud computing and cloud architectures as well as on geospatial data processing and large spatial databases in the past, the combination of the two, cloud architectures for spatial processing, has only become subject to research in the last couple of years [2, 11]. The availability of commercial cloud solutions such as Amazon EC2 or Microsoft Azure has facilitated applications in this area. For example, Qazi et al. describe a software architecture for Modelling Domestic Wastewater Treatment Solutions in Ireland [5]. Their solution is based on the Amazon cloud services on which they install the commercial tool ArcGIS Server via special Amazon Machine Images (AMIs) provided by ESRI. Qazi et al. make use of ArcGIS Server's REST interface to deploy web services providing the 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 to that, their work depends on the commercial ArcGIS Server and the respective Amazon Machine Images. Our architecture, on the other hand, makes use of freely available open-source software and can be configured to run on different infrastructures.

Li et al. on the other hand leverage the Microsoft Azure infrastructure to process large volume datasets of satellite imagery in a short amount of time [3]. 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 [13]. However, it was explicitly developed for the Azure API which provides a queue-based task model that is 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. ESRI GIS Tools for Hadoop, for example, provides 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. While there has been work leveraging this framework [16, 17] 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 even a lot faster [18]. 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.

Since geospatial applications in the cloud are quite new, the community is still looking for best practices. Agarwal and Prasad report from their experience with implementing a cloud-based system called Crayons that facilitates high-performance spatial processing on the Microsoft Azure infrastructure [11]. 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's functionality. Our architecture is service-oriented and consists of loosely coupled components that can be exchanged quite easily. That way, a wide range of spatial processing services are supported and can be extended later without requiring fundamental changes to the architecture. Additionally, our approach allows individual components to be replaced if requirements should change in the future.

The OGC (Open Geospatial Consortium) has recently set up a new domain working group for Big Data dealing with service-oriented architectures for distributed processing of spatial data. Our architecture is very flexible and allows for various kinds of processing services. This also includes web processing services such as the OGC WPS (Web Processing Service). 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. This imposes a major performance hit. In our architecture we deploy a distributed file system (see section 7) to which processing service are directly connected. This allows for 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 very beneficial if this improves the quality of processing results.

Our system employs a workflow editor (section 5) and a component called job manager (section 9.3) that is able to manage the cloud infrastructure, to deploy services to computing nodes and to execute distributed workflows. There are other approaches that also support cloud-based workflow management. Malawski et al. describe a model for the execution of scientific applications [12]. Based on their experience with grid-based

component models they show how to dynamically couple so-called workers to create a processing workflow. These workers are stored as a ready to deploy virtual appliance and provide a standardised interface to describe how to connect it to other components. A Ruby-based API allows developers and scientists to create workflows by specifying all involved workers and their order of execution.

The Pegasus Workflow Management System [19] is similar to the approach of Malawski et al. in the sense that it also provides a programming interface for developers and scientists using Java, Python or Perl. Workflows are specified by describing a DAG (Directed Acyclic Graph) in code. Pegasus automatically translates this graph to an executable workflow that can run on various backends (i.e. cloud and grid infrastructures) such as Amazon EC2, Nimbus, or Open Science Grid.

Compared to Pegasus and the approach from Malawski et al., our architecture, on the other hand, allows the user to define a workflow using a high level DSL which reduces the required in-depth knowledge about existing components, execution order or a particular programming language. Instead of a common interface we rely on metadata describing each processing component and, in particular, how to deploy and execute it to simplify the migration of existing components to our architecture. Finally, we eliminate the overhead of managing virtual appliances by putting our job manager in responsibility of creating compute nodes with all needed libraries and the processing component itself.

Note that there are other approaches to workflow modelling, mostly graphical ones such as Activiti[1], Visual Paradigm[2], or the Workflow Editor from the Java Workflow Tooling (JWT)[3]. These tools focus on Business Process Modelling (BPM) but can be used for other purposes as well. The Pegasus Workflow Management System even contains a graphical editor where users can draw workflow diagrams directly without knowledge of any programming language.

While it is often easier for people to read a workflow in a graphical way—using boxes that represent actions and arrows to depict the data flow—it can be hard for them to create their own designs. Whitley reports that untrained people often find it hard to work with graphical editors as diagrams can quickly become very complex [20]. The readability of any representation (no matter if graphical or textual) depends on the so-called secondary syntax, meaning the way things are organised on the screen. Untrained people often struggle with placing diagram elements (boxes and arrows) in a meaningful manner.

However, flow diagrams, for example, are typically well suited to represent workflows and the tools above also assist users in structuring. On the other hand, compared to our domain-specific language, existing workflow tools are rather generic. A DSL can be adapted to a specific use case and domain vocabulary. The workflow scripts in our case also support a declarative modelling style and are hence often very short (see examples in section 5 or 13). Users don't have to specify how exactly the system should perform a task. The interpreter (section 9.2) and job manager (section 9.3) figure this out based on pre-defined rules. Such a declarative modelling style is supported by existing graphical solutions only to a certain extent.

[1] http://activiti.org/

[2] http://www.visual-paradigm.com/

[3] http://www.eclipse.org/jwt/

3. Requirements

The IQmulus research project aims for creating a platform for the fusion and analysis of high-volume geospatial data such as point clouds, coverages and volumetric data sets. The architecture design presented here is based on a detailed requirements analysis that has been carried out in the IQmulus project together with users from the land, urban and marine domain. Based on that, we designed mock-ups for the user interface and created robustness diagrams, a technique known from use-case driven object modelling [21]. The robustness diagrams helped us to identify components needed in the architecture. Such an approach makes sure a large number of user requirements are met and that the system generally provides the desired functionality. The results of this analysis can be found in the public deliverables D1.2.1, D1.2.2, D1.2.3 as well as D2.3.1 of the research project IQmulus [22–25].

Besides the identified requirements there are general aspects related to the processing of large-volume geospatial data as well as sustainable software architectures. The issues that we considered for our architecture design are presented in the following subsections.

3.1. Big Geo Data

According to Laney there are three aspects defining *Big Data*: volume, velocity, and variety [26]. Applied to geospatial processing (*Big Geo Data*) we can conclude the following:

Volume. As described above geospatial data sets tend to be very large. In particular if modern sensor networks, LiDAR scanners (terrestrial as well as aerial), and artificial satellites are involved that produce several GiB or TiB per hour. A system that processes such amounts of geospatial data should be scalable, so it is able to process an arbitrarily large data volume. Our architecture makes use of mature components (e.g. the Apache Hadoop File System HDFS) that have been proven to be scalable. Additionally, we use a configurable rule set that allows our components to adapt to the cloud infrastructure and, hence, the available resources.

Velocity. Modern sensors produce large data streams in a very short amount of time. The faster the data can be processed by a system the higher the derived value of information. For example, today aerial images, orthophotos, and digital terrain models are extensively used for urban planning, but due to lack of time and money they are only updated every second or third year [27]. An automated system that is able to generate this information from a very large input data set (i.e. satellite imagery and LiDAR point clouds) in a short amount of time can help urban planners consider current developments. Our system allows for such automated processes through user-defined workflows (see section 5). In addition to that, it is rule-based and users can specify if they prefer speed over accuracy or vice-versa (see section 9.3).

Variety. Geospatial data is typically very heterogeneous. Oftentimes data from different sources has to be combined, multiple data exchange formats, reference systems, and accuracies have to be considered, etc. A system processing heterogeneous geospatial data should be very flexible to accommodate for this. The architecture presented here meets this requirement as it consists of small atomic and independent processing services that can be flexibly chained to create a larger workflow.

We see two areas where the cloud offers reasonable benefits over traditional geospatial information systems.

High-performance computing. The cloud offers virtually unlimited resources in terms of processing power and memory. Cloud infrastructures are scalable and elastic which means new resources (more processing nodes or more memory) can be acquired on demand. As mentioned above, the faster geospatial data can be processed the higher is its practical value.

Distributed services. In addition to that, the cloud, and in particular a Software as a Service solution (SaaS), offers distributed services that can be used "to access, process, visualise, and share data, metadata and models from various domains for various purposes" [28]. Access to cross-thematic and linked geospatial information helps GIS experts in planning and decision making.

In this work we mainly focus on the high-performance computing aspect. Although our system also offers distributed storage we have not investigated this area thoroughly yet.

3.2. Sustainable software architectures

There are a number of articles that describe quality indicators for sustainable software architecture design as well as architecture evaluation methods [29–32]. In this work we focus on the following aspects.

Functionality. As described above, a detailed requirements analysis has been carried out before the architecture design took place. The solution presented here and the functionality it provides reflects the identified requirements.

Performance. Our architecture employs components that make use of rules specifying declarative and procedural knowledge to decide in which way geospatial processes should be run in the cloud to make best use of existing resources (see section 9.3). Moreover, the geospatial processes are designed to be highly scalable using multi-core algorithms, MapReduce, or similar techniques.

Availability. Our system is expected to be able to perform well in various situations. For example, in a catastrophe scenario it should deliver results in a short time and with a high degree of reliability. The components in the system are therefore designed to be redundant, so there is no single point of failure (SPOF). The distributed file system, for example, offers a high fault tolerance in respect to hardware errors through data replication. Moreover, we use NoSQL databases that are designed to run in the cloud and offer high availability features such as replication and automatic failover.

Modifiability. The architecture is service-oriented. Loose coupling and high cohesion make sure services can be interchanged and connected in different ways. The fact that components are independent from each other allows the architecture to be modified quite easily later on without requiring the specific service implementations to be changed.

Portability. The architecture does not require a specific hardware or operating system. This is rather important for a distributed system as the individual components may run on different platforms and machines. We rely on open-source components that are compatible to various platforms. We also provide a generic interface for data access without requiring the components to handle platform-specific issues (see section 8).

Configurability. Our system allows end users to define high-level processing workflows according to their needs. These workflows are written in a domain-specific language (DSL), a programming language tailored to a specific application domain. Since the DSL consists of domain vocabulary the workflow definitions are easily readable by experts working in this domain. At the same time the language has a defined syntax and grammar and is hence parsable by machines. The DSL allows end users to specify constraints such as the desired output quality (accuracy, completeness, etc.) or the kind of algorithms to be used (fast vs. precise algorithms). Our system uses rules to reason about these constraints and to produce a processing chain that matches them best.

4. Overall architecture

In this section we present the overall architecture and give a short description of all components and how they work together (see also figure 1 on the following page). In subsequent sections we describe the main components in detail.

A 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 the workflow editor.

First the GIS expert uses the data upload form to store new geospatial data together with related metadata (c.f. section 10) in the cloud. The upload form sends the data to the data access service (section 8) which saves it in the distributed file system (section 7). Metadata can either be entered manually in the upload form or provided as an ISO 19115 compliant XML file. Existing data sets can be accessed through the data browser which allows for searching the file system based on a spatial extent or metadata.

The GIS expert then specifies a high-level workflow using a domain-specific language (section 5). The workflow is saved in a workflow database for later use. Additionally, it can be shared with other users who have similar requirements.

Next, the GIS user executes the workflow through the user interface and selects whether the system should prefer processing speed over accuracy or vice-versa. The workflow will first be parsed (section 9.1), interpreted (section 9.2), and finally processed by the job manager (section 9.3) which queries the catalogue service (section 10) for metadata about processing services and the data to be processed. The job manager applies pre-defined rules to create a process chain specifying which processing services (section 6) should be executed in which order and on what nodes in the cloud. The job manager starts the services and monitors their execution while the services store their processing results in the distributed file system.

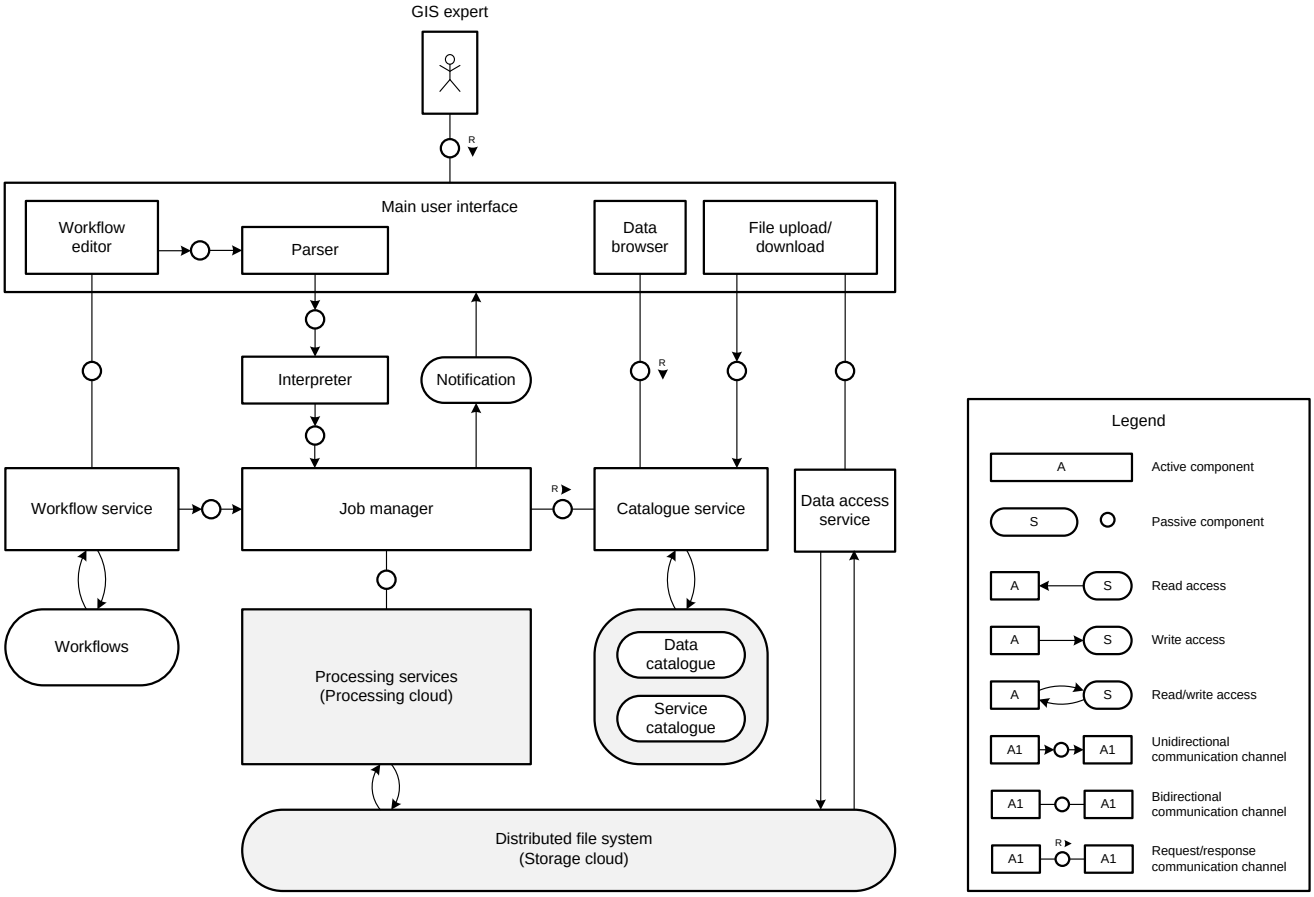

Figure 1: The overall architecture of our system

Finally, the job manager 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.

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 has a number of features that help users create the workflow—for example, syntax highlighting, auto-completion, and context-sensitive cue cards. Workflows are specified using a domain-specific language (see screenshot in figure 2 on the next page).

DSLs have been used before in the area of cloud computing to control distributed processing. For example, Apache Pig, a platform for analysing Big Data sets, provides a language called Pig Latin [33]. It 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. The language drives distributed MapReduce jobs [13] executed in the cloud by the Apache Hadoop framework. Pig Latin therefore simplifies the process of specifying complex parallel computing tasks which can sometimes be tedious even for experienced programmers. However, it is very generic and lacks support for geospatial processing. This gap is closed by the SpatialHadoop framework [16] which adds spatial indexes, geometrical data types and operations to Hadoop. In addition to that, SpatialHadoop offers a DSL based on Pig Latin. Pigeon is a query language that allows users to specify complex spatial queries in a readable and concise way [34].

Our workflow editor goes a bit further and provides a *high-level* DSL that does not require users to know details about available processing services, 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 from the distributed file system 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
    add to CityModel
end
```

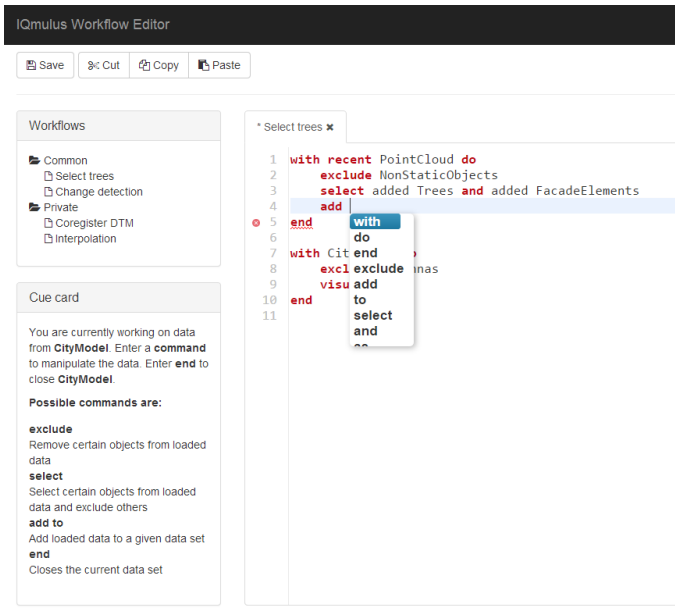

Figure 2: The workflow editor assists users in writing high-level geospatial processes. It supports syntax highlighting and auto-completion. In addition to that, context-sensitive cue cards on the left tell users about their options.

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 9.2).

The workflow finally removes all *Antennas* from the city model and visualises the result.

```
with CityModel do
    exclude Antennas
    visualize
end
```

For more information about the DSL used here and the language design process we refer to our previous work [27].

6. Processing services

Geospatial data is typically heterogeneous. GIS experts often need to combine different kinds of information such as vector data, point clouds, or aerial images to one single data set. Our system has to be able to handle various data exchange formats, different resolutions, spatial reference systems, etc. Supporting a wide range of spatial algorithms is key to a system that can be used in multiple use cases.

The spatial algorithms in our system are implemented by experts from various domains with different backgrounds—e.g. mathematics, photogrammetry, or land survey. Each algorithm is provided as a separate *program*—we call them *processing services*. Linking multiple processing services to a chain allows for creating complex workflows. A similar approach can be found in the Unix operating system where pipes can be used to send data through multiple programs. Just like in Unix each of our processing services serves exactly one specific purpose.

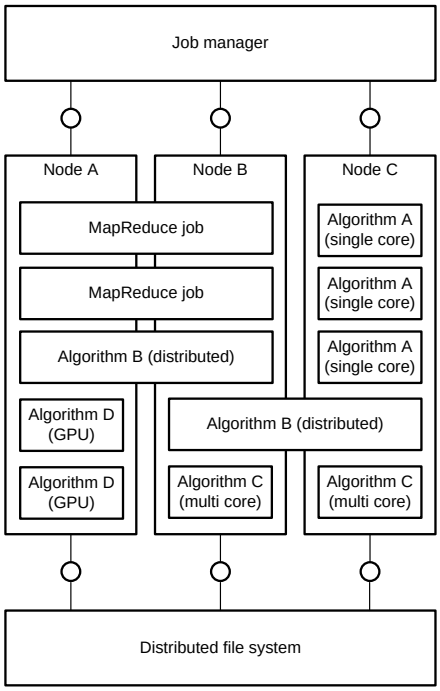

Figure 3: An example of different processing services that can be deployed to the cloud. The job manager oversees their execution whereas the distributed file system is the main communication channel for all processes.

This allows for a very high flexibility. For example, there is a service for corregistration, one for triangulation, one for intersecting 2D or 3D data, etc. Input and output parameters of each service are described in a catalogue (see section 10) so the system knows how they can be connected (this process is described in section 9.3).

The processing services have been developed using various programming paradigms. As described above, while MapReduce is currently one of the hot topics in cloud computing 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. Furthermore, 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.

Our architecture therefore supports the following types of algorithms (depicted in figure 3).

MapReduce jobs. We use Apache Hadoop to execute 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.

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 job manager executes them and oversees their resource usage.

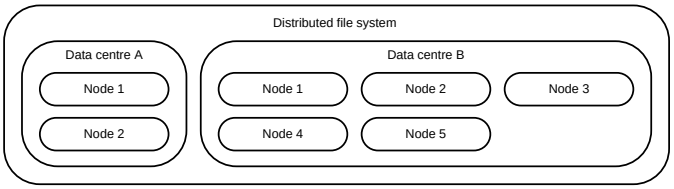

Figure 4: An example depicting how multiple nodes from different locations (i.e. data centres) contribute to the distributed file system.

Multi-core/GPU algorithms. Such algorithms run on a single node only but use multiple CPU or GPU cores in order to increase performance. Those algorithms 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 job manager distributes input data to multiple instances of such algorithms if possible (see details in section 9.3).

Single-core algorithms. Our architecture allows single-core algorithms (most likely legacy algorithms) to be executed. They are treated like multi-core algorithms but in order to parallelise the processing the job manager has to split input data and distribute it to several instances of these services (see section 9.3).

7. 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 4). This means that the individual nodes may use different operating systems and different file systems but the DFS provides a common interface for applications to access data on these nodes. A distributed file system typically has the following capabilities.

Location Transparency. The application can access the data as if it was stored locally no matter where it is actually located—i.e. on which node and in which data centre.

Access Transparency. The DFS provides a common interface for applications to access data in a consistent way, independent of the underlying operating system and file system.

Fault-tolerance. A DFS usually replicates data to a number of nodes. If one node fails the data will still be available on other ones.

Scalability. A DFS can operate on a small number of nodes up to a large number. New nodes can be added on demand and, more importantly, without any downtime.

The distributed file system in our architecture stores uploaded data and processing results. It also acts as the main communication channel between the processing services. If multiple processes are called sequentially or in parallel all intermediate results will be transferred through the DFS. We chose this approach over direct communication between the services to provide a high modularity of the system. It allows processing services to run independently, asynchronously and in different orders to enable a wide range of processing workflows. Independent processes may be replicated and transferred to other nodes in the cloud on demand and be executed there. This only works if these nodes provide enough transparency in terms of location and access—capabilities which are provided by the DFS. First, the processing services themselves do not need to care about where the data is located. Second, they can use a common interface which does not depend on the actual technology used to store the data. By applying these principles the underlying file systems or operating systems and, therefore, the underlying nodes can easily be interchanged without affecting the way the processing services work.

There are a lot of scenarios where analysing and processing geospatial data is not only critical in terms of time but in terms of availability. For example in a disaster scenario it is mandatory to get fast and accurate results to provide good decision support. The scalability offered by a DFS allows the number of nodes involved in the processing—and therefore the computational power—to be increased on demand whereas fault-tolerance guarantees the availability of all needed data. The latter is accomplished by replicating the data to different locations in terms of data centres, countries or even continents.

The Hadoop Distributed File System (HDFS) is open-source released under the Apache license. Apart from the general properties described above, HDFS is specifically designed for large data sets and high availability. Due to the fact that the file system is implemented in Java it is also platform-independent. The HDFS API allows applications to access files just as if they resided on any other file system. In fact, even the local file system can be accessed through the API.

Depardon et al. compared several distributed file systems such as HDFS, iRODS and Lustre [35]. They have shown, that the main difference is not about performance but about the design of the file system. HDFS is the only DFS they compared that offers automatic load balancing. The fairly good performance in combination with its platform independence and the sophisticated MapReduce support make it a good choice for our architecture

Apache Spark, an open-source library for high-performance data processing [36], supports HDFS and offers an API for various programming languages including Java, Scala and Python. This allows many geospatial processing applications to access the file system.

Applications written in C, however, require a workaround. The C API for HDFS based on the Java Native Interface (JNI) is not well supported. We therefore set up an NFS (Network File System) wrapper service that provides a virtual, mountable file system but internally forwards to the HDFS. This approach implies overhead and can lead to a performance drop if the application tries to access a file randomly instead of sequentially. HDFS is stream-based and random access has to be simulated by sequential reads. We therefore suggest that such legacy applications—although our architecture supports them—should be avoided or reimplemented, so they access the HDFS directly.

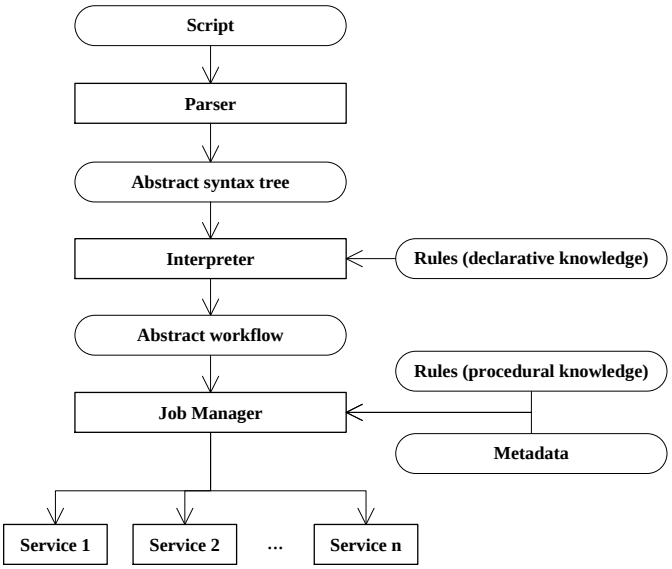

Figure 5: Scripts are parsed to an abstract syntax tree. An interpreter traverses this tree and generates an abstract model that is independent of the DSL. According to this model the job manager than executes the processing services in the cloud.

8. Data Access Service

The Data Access Service provides a RESTful HTTP-based interface to the Hadoop Distributed File System (HDFS) and offers operations to upload, read and delete files or directories as well as to list files in directories, set permissions etc.

Several, redundant instances of the data access service may run in the same cloud. Since the service itself is stateless, the instances do not have to communicate with each other. However, they all access the same distributed file system and therefore serve the same content.

Being a REST service the data access service provides a couple of handy features such as different resource representations (e.g. HTML or JSON) and a self-descriptive interface. In particular, the latter means that clients can browse the file system by entering it at the root level (the service's main entry point) and then following hypermedia links. This results in a higher level of abstraction and a flexible interface, where changes applied to the interface won't break clients—c.f. the HATEOAS constraint (Hypermedia As The Engine Of Application State) which is an essential part of REST [37].

9. Executing workflows in the cloud

In order to execute the workflows defined by the GIS experts in the workflow editor (section 5) they first have to be parsed and then interpreted. The following sections describe this process which is split into three phases (c.f. figure 5) handled by different components: the parser, the interpreter, and the job manager.

9.1. Parser

The domain-specific language used in our workflow editor is specified through a Parsing Expression Grammar (PEG). In fact, programming languages can as well be defined in a context-free grammar (CFG) using EBNF (Extended Backus-Naur Form). One of the benefits of PEGs, however, 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 don't 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.

We use the open-source library PEG.js[4] to automatically generate a language parser that can be embedded in our user interface running in the web browser. The main purpose of the parser is to create a machine-readable model of the workflow—i.e. an abstract syntax tree (AST)—which can be traversed by the interpreter.

9.2. Interpreter

The interpreter traverses the AST generated by the parser and creates a model that is independent of the domain-specific language. It translates terms in the workflow to processing services or processing parameters. The interpreter makes use of *declarative knowledge* encoded in pre-defined mapping rules. In their book 'Model Driven Engineering and Ontology Development' Gaševic at al. define declarative knowledge as follows [38, p. 12].

> Declarative knowledge describes what is known about a topic or about a problem. For example, some statements of declarative knowledge may describe details of concepts and objects.

Gaševic at al. specify that declarative knowledge consists of concepts, objects, and facts. In our case we use this kind of knowledge to express relations between terms in the workflow DSL and processing services—including parameters, if necessary. Terms in the DSL are actually instances (i.e. objects) of concepts taken from the application domain. The declarative knowledge expresses what we know about them, namely how they map to processing services and parameters. The declarative rules help the interpreter transform the abstract syntax tree to a model that is independent of the domain-specific language.

Figure 6 shows mappings that appear in the example workflow from section 5 above.

- **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.** The terms `recent` and `PointCloud`, for example, are mapped to a processing service that searches the distributed file system for a data set containing the most recent version of a point cloud.

[4] `http://pegjs.org/`

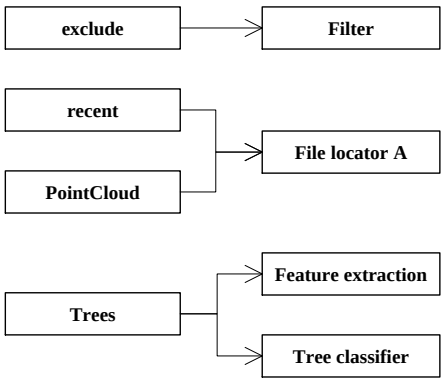

Figure 6: Example of how terms in the abstract syntax tree can be mapped to data sets and processing services.

- **One-to-many.** Terms such as `Trees` may 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 as well as a pre-trained classifier for trees.

In addition to that, **many-to-many** relations can also happen although they do not appear in the example.

By the use of mapping rules as it is proposed here the interpreter may be replaced without affecting the DSL and the parser. This means that even if the back-end (i.e. the interpreter, the mapping rules, the job manager, and the processing services) are replaced by other implementations the scripts written by the domain users stay the same. In particular, this ensures the domain knowledge that 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.

Note that, according to the overall architecture (figure 1) the interpreter does not have access to the distributed file system or the catalogue service. This means that the interpreter knows little about the actual files stored in the cloud, the input and output parameters of the processing services, or the actual infrastructure the workflow is executed on. All knowledge required by the interpreter has to be specified in its declarative mapping rules. Based on this, the interpreter generates instructions for the job manager which apparently has access to more information and can therefore generate a concrete process chain.

9.3. Job manager

The job manager traverses the abstract model generated by the interpreter and generates a concrete process chain that specifies which services have to be executed in what order.

The catalogue service (see section 10) provides further information about the services and the data to be processed. This includes input/output parameters, accuracy, completeness, etc. The job manager uses procedural knowledge encoded in production rules to evaluate the workflow and the additional data. Depending on these production rules and on the constraints defined with the workflow the job manager may automatically allocate resources in the cloud. For example, a typical production rule looks like as follows (pseudo code).

```
when
    user needs results as fast as possible
then
    increase number of processing nodes
```

The job manager tries to meet this constraint by replicating more virtual machines in the cloud and spawning more processes.

If users indicate they prefer performance over quality the job manager may select a processing service that is faster but produces less accurate results, or select a data set that is smaller and hence faster to process but not as complete or detailed as others.

```
when
    user prefers speed over accuracy
then
    use terrain grid instead of LiDAR point
        cloud
```

The job manager may also produce a process chain that suits the infrastructure it should be executed on. For example, the following rule affects on which node a certain service is deployed.

```
when
    processing service X needs Windows
then
    deploy X on Windows node
```

Additionally, the job manager can handle resource management issues.

```
when
    data is too large for a single node
then
    split data into tiles
    distribute tiles to multiple nodes
```

When handling high volume data, the job manager takes also care of deploying the services close to the data to be processed e.g. the same data centre. This is meant to decrease the amount of transferred data over a broadband connection, hence increasing the performance of the whole workflow.

We developed the job manager using the rule-based system Drools[5]. This open-source framework has been implemented in Java for quite some time already and is very mature with a large community. Additionally, Drools offers a declaration language called the Drools Rule Language (DRL) to simplify the process of defining the rules themselves. Other rule-based systems use more complex notations. For example, in JESS rules have to be defined in LISP. This makes the rule definition process, in our experience, more complex and error-prone.

The rules are stored in a database. This allows us to add, remove or change them in production without downtime. Additionally, it allows us to deploy multiple instances of the rule system, while all use the same set of rules. Hence we are able

[5] `http://www.drools.org/`

to increase the availability and scalability of the job manager significantly.

We use the NoSQL database MongoDB which is tailored to be deployed to the cloud. We prefer MongoDB over a classical, relational database as it offers advanced features such as replication and automatic failover strategies out of the box. The set of rules is a crucial part of the job manager. Hence its unavailability would lead into the inability to reason about the workflow. MongoDB's replication feature allows us to store the data redundantly, so there is no single point of failure. Note that even though we chose MongoDB here, other cloud-capable NoSQL databases such as CouchDB, for example, may also be used. The loose coupling of our architecture allows us to easily exchange individual technologies. In our experience, however, MongoDB is easy to install and use and meets our requirements very well.

9.4. Parallelisation

The job manager is responsible for distributing the processes to the different nodes in the cloud. In order to make best use of the available hardware, individual processes can be implemented with parallel, multi-core or distributed algorithms. Some processes might use algorithms that utilise GPU cores (if available) to perform high-performance calculations or use MapReduce. The job manager has to take care to choose the proper infrastructure—i.e. GPGPU cluster or Apache Hadoop. Furthermore, there are existing algorithms which only use a single CPU core. In this case the job manager needs to find a good strategy to split the input data and distribute it to multiple instances of this service. To summarise, the job manager needs to handle the following cases (c.f. the list of algorithm types in section 6).

MapReduce jobs. The job manager forwards MapReduce jobs to Apache Hadoop. Hadoop is responsible for executing the job and for distributing work to the different nodes in the cloud. The job manager configures the infrastructure and monitors the process execution.

Distributed algorithms. Processes can be implemented with frameworks for distributed computing such as agent-based frameworks or Message Passing Interface (MPI). The job manager has to keep track of which nodes these processes run on.

Multi-core/GPU algorithms. The job manager has to keep track of used CPU/GPU cores in order to decide how many processes can run on one node at the same time.

Single-core algorithms. The job manager has to find a strategy to split input data and to distribute the chunks to the various processing services. Also, the job manager is responsible for spawning the single-core processes and to distribute them to the different nodes in a way that best utilises the possibilities of the available hardware.

Again, the job manager makes its decisions based on production rules. Let us suppose there is an arbitrary single-core service A, and furthermore a service B which is able to split input data into tiles. The following production rule can be used to describe the dependency between these two services (pseudo code).

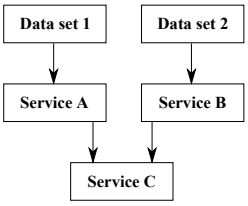

Figure 7: A processing chain with two independent processes A and B and a process C that can only run if results of A and B are available.

```
when
    processing service A is called
then
    call processing service B first and
    distribute results of B to multiple
        instances of A
```

The generated process chain may not only contain one of the described process types but any combination as depicted in figure 3 on page 6. The job manager will take this into account and consequently start the next process in the chain as soon as possible. In particular this means that it can start instances of a processing service even if the former step has not returned its full result set. Obviously it is necessary that intermediate data will be consistent enough to fulfil the requirements of at least one instance of the subsequent processing service.

In addition to that, the job manager can execute services that are completely unrelated to each other in parallel. For example, figure 7 shows a processing chain where service C depends on the results of A and B. Obviously, A and B can be executed in parallel since they are independent. Execution of C may only start if the results of A and B are available.

10. Catalogues

Geospatial data is usually stored along with metadata providing additional information such as resolution, accuracy, completeness, etc. Metadata is needed to interpret and process data in a proper and meaningful way. The job manager makes heavy use of this data for decision making as described in section 9.3. Additionally, it also needs information about the processing services such as input parameters, algorithmic accuracy, etc.

Since the job manager cannot work properly if the metadata is not available, we use MongoDB to store these two types of metadata (as described in section 9.3). MongoDB provides replication and automatic failover and recovery strategies that allow us to set up a highly available system.

The metadata is provided by two independent services acting in front of the database. The data catalogue and the service catalogue provide access to the geospatial and processing service metadata respectively. They offer a REST interface which can be used by any component that needs access to the data without knowledge about the actual database in use.

10.1. Data catalogue

The Data Catalogue Service provides access to all metadata stored along with the geospatial data residing in the distributed

file system. This catalogue is designed to support metadata standards such as ISO 19115 and 19119. The job manager primarily uses data such as resolution or size to distinguish datasets which are fast to process from others which are very detailed. It also uses this data to decide how to split and distribute them to different instances of one service. If necessary the processing services may use this information as well.

10.2. Service catalogue

Every service which will be deployed to the system comes with a file providing information about the service and how it can be executed in the infrastructure. The service catalogue observes these files, caches the content and provides access in a uniform way. The job manager requires the metadata listed in table 1 to build the process chain, prepare the infrastructure, and deploy the services. Note that the job manager solely acts based on the pre-defined procedural rules. It has no additional a priori knowledge. It must be possible to completely infer the knowledge about how two services can be connected to each other from the rules in combination with the service metadata.

Metadata	Description
ID	Unique identifier to be used while defining workflows
URI	URI of Service (may include placeholders e.g. for parameters)
input	Representation and format of data
parameters	Applicable parameters
output	Representation and format of data
language	The programming language used for the implementation
libraries	Libraries the service depends on
OS	The operating system the service relies on
accuracy	Algorithmic accuracy
computation time	Computational time in relation to data size
locality	If the algorithm is applicable to a subspace of the data
alternatives	Services which provide same functionality but use different algorithms

Table 1: Relevant service metadata for process chain generation

11. Implementation challenges

The architecture presented here has been developed for the IQmulus research project which is dealing with high-volume fusion of geospatial data. While implementing our solution we were facing a number of challenges. For example, designing the domain-specific language for the workflow editor was a complex task. Based on our previous work [27] we used a DSL modelling method which consists of several steps, including the text-based analysis of all user stories gathered in the user requirements analysis phase, as well as the definition of a domain model from which we could derive the terms needed for the DSL. The main challenge was to create a language that is on the one hand powerful enough to cover the user requirements, but on the other hand clear enough to be understandable for the domain experts. Close communication with the users is in our opinion key to successful DSL design.

Another challenge was the infrastructure management—i.e. provisioning of new nodes in the cloud, resource management, etc. We were investigating several solutions and finally decided to rely on the open-source software OpenStack. This tool allows us to manage multiple cloud infrastructures (e.g. Amazon EC2 or VMware vSphere) and therefore keep our architecture compatible to various platforms. In addition to that, we had to find a solution for the automated provisioning of the processing services as well as the components of our architecture to the nodes in the cloud. After testing several tools (Puppet[6], Chef[7], etc.) we decided to use Ansible[8] which is a lightweight open-source solution that is able to execute provisioning scripts facilitated by a role-based node management framework.

One of the major challenges was the definition of a mapping from terms in the DSL to processing services as described in section 9.2. Our aim was to decouple the workflow language from the actual execution in order to be able to change or replace either one without affecting the other. In other words we wanted to be able to execute the same workflows on different cloud infrastructures. Additionally, it should be possible to replace the DSL (if necessary) without requiring us to change the processing services. The rule-based approach that we have chosen allows us to dynamically change the mapping and to adapt the system to different use cases and scenarios.

One problem that is yet to be solved is the efficient scheduling of geospatial processes in the cloud. The production rules the job manager uses to decide on which node a service should be executed under certain circumstances are in our opinion key to solve this issue. They allow the system to be configured and adapted to various platforms and use cases. However, in order to execute the workflow as efficiently as possible and to make best use of available resources, it is often hard to decide in advance on which node and on how many nodes processing services should be installed. In a typical cloud environment there is very little information about the performance of each node. Even though the number of CPU cores and the available main memory is known, the cloud infrastructure provider may assign physical resources dynamically amongst a number of applications running on the same cloud. Processing time of the same workflow may therefore vary between multiple runs. In addition to that, although there is information about the performance of processing services (e.g. their order of complexity) it is still hard to tell in advance how the algorithms perform on a specific hardware or platform.

Since the job manager uses production rules for its decisions we may be able to let it learn from previous workflow executions. The production rule base could be extended dynamically during runtime and so the job manager's assumptions

[6] `http://puppetlabs.com`
[7] `https://www.chef.io`
[8] `http://www.ansible.com`

could be underpinned by experience from the past. However, we have not yet implemented this approach and so the issue remains subject to further research.

12. Security considerations

In order to protect the rights of owners of geospatial data stored and processed by our system various security issues have to be considered. This section gives an overview over 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 7 we use the Hadoop file system (HDFS) as the means to store geospatial data redundantly and in a distributed manner in the cloud. HDFS does 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 or, since HDFS is open source, data encryption could be implemented directly into the file system. Note that in a private and trusted cloud environment data encryption typically has no benefits, but imposes performance and development overhead instead.

Data transfer. In a distributed system data is typically transferred between multiple nodes. Oftentimes it even has to leave a data centre and has to be copied to another one. It is important that data is encrypted before it is transferred. As HDFS is our main communication channel, we have to make sure all communication channels used by the HDFS name node and data nodes are secured. This can be enabled in the HDFS configuration file.

Data upload and download. The data access service (see section 8) is a REST service that enables access to the data stored in the distributed file system through a specified HTTP interface (either from a client application or through the Web Browser). Access to this service is secured through HTTPS using SSL/TLS encryption.

Authentication. We use single sign-on authentication facilitated by CAS (Central Authentication Service), an open source framework that allows users to authenticate using common credentials for various components of the system. In our architecture user credentials are stored in a distributed MongoDB database. Passwords are never stored as clear text but hashed using the Bcrypt hash algorithm which is typically considered very secure [39].

Authorisation. While authentication makes sure the right person accesses the system, it does not protect data stored in the distributed file system against unauthorised access. HDFS implements a sophisticated permission model for files and directories that is very similar to the one found in the UNIX operating system. Each file is associated with an owner and a group. Separate read/write permission 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.

13. Evaluation

In the IQmulus project three showcases from the marine, land, and urban domain have been defined consisting of eleven use cases (i.e. processing workflows). In this section we will present the results of applying our architecture to two of these use cases coming from the urban and land domains.

The first workflow deals with individual tree extraction from urban LMMS data (Land-based Mobile Mapping System). The test data set used is a large point cloud split up into 339 files with a total size of 176 GB. Each file has exactly 3 million points (except for the last file which contains the remaining points).

The workflow expressed in our domain-specific language is very short and looks as follows:

```
with each PointCloud do
    extract Trees
end
```

The interpreter maps the `extract` keyword to a processing service performing multi-object classification for 3D point clouds. The service depends on the results of another one calculating the Point Cloud Dimensionality (PCD). The result of the multi-object classification is a new point cloud annotated with labels indicating point classes.

The interpreter also maps the term `Trees` to a service identifying individual trees based on a classified point cloud. It generates the abstract workflow and passes it to the Job Manager. The Job Manager searches the distributed file system for point clouds (term `each PointCloud` in the DSL). It then calculates the dependencies between the involved services and executes them in parallel on multiple computing nodes.

Note that the DSL script is very short but powerful. The reason for this is two-fold. First of all, the rules used by the job manager define dependencies, so even though only one task has been specified in the workflow, the job manager will execute three services. Second, the services have default parameters not specified in the DSL script. Of course, experienced users may override these parameters (see the second example below).

While running this workflow we found the data access service to be very valuable to upload large data to the distributed file system. Using a web-based user interface is certainly easier than Hadoop's command line tools.

Concerning processing performance we were able to reproduce the results we achieved on a high-end workstation. This machine had a similar configuration as a virtual node in our cloud. Each file took about six minutes to process. This suggests that virtualisation overhead in the cloud is negligible—an observation also reported by Malawski et al. [40]. Nevertheless, compared to the single machine, the cloud performed very well because we were able to distribute services to multiple nodes and therefore process many tiles at once. Scalability in this scenario is theoretically linear which means that the more computing nodes involved the better the performance. In practice, however, it depends on the capabilities and capacity of the underlying infrastructure and hardware. The number of real virtual machine hypervisors and CPU cores as well as the available amount of physical memory limit the virtual computation

power. In our experiment we used ten virtual machines running on six hypervisors with a total number of 96 cpu cores, and we didn't notice any significant performance degradation.

The second workflow that we used for evaluation deals with flood and water detection in a wide area (i.e. on the level of a country). The data under test is a set of pre-processed, geo-referenced satellite images stored in GeoTIFF format. Before they can be used in the actual processing the images need to be enhanced in terms of contrast, saturation, etc. There are eight images in total, each having a size of 250 MB.

The workflow as specified in the DSL workflow editor is as follows:

```
with each SatelliteImage do
    enhance image using {factor: 2.0,
        offset: 0.0,
        method: toaReflectance}
    detect FloodedAreas
end
```

The workflow employs three services. The term `enhance` in the DSL script is mapped to a service performing radiometric enhancements based on the given parameters. The term `detect` maps to two services which firstly compute spectral indices on the given satellite imagery and finally detect flooded areas and waterlogging.

The example is executed in the same way as the first one. The interpreter maps the terms in the DSL script to processing services. The Job Manager looks for the requested imagery and starts the processing services in parallel. This time it can make use of the capabilities of Apache Hadoop as the algorithms are implemented in MapReduce.

Again, the performance in the cloud is comparable to the one on a high-end workstation with a similar configuration. The three services take about 3 to 4 minutes to process one file. However, the cloud allows all files to be processed at once on 8 different nodes in about 4 minutes which is not possible on a local machine.

There are some issues to be considered though. First of all, debugging the processing algorithms in the cloud is rather tedious. Developers have to build services, upload them to the cloud (i.e. register them with the Job Manager), run the test, download the log files to see what has happened, fix the bug, and then start all over again. This can be quite time-consuming but while the services become more mature they don't have to be updated as often anymore.

14. Conclusion

In this paper we presented a modular architecture for processing of Big Geo Data in the cloud. We described our overall architecture, its components and how they work with each other. We also described requirements for systems that process Big Geo Data and how we address them in our architecture.

Compared to previous work we did not focus on only one specific programming paradigm such as MapReduce for geospatial processing. Our architecture is modular and allows for executing a wide range of processing services (even legacy ones). This makes the system suitable for many applications. In addition to that, we did not focus on a certain cloud infrastructure. Instead our solution makes use of a rule-based system that allows us to control the execution of workflows on multiple platforms.

Further flexibility is given by the fact that our architecture allows domain experts to control the processing of Big Geo Data through a domain-specific language. This makes the cloud and therefore its processing power available to users from the geospatial domain without requiring them to have a deep knowledge of the infrastructure and its technical details.

The evaluation results presented in section 13 suggest that our architecture works well. The DSL is very powerful and allows domain experts to control processing in the cloud with only a couple of commands. The performance gain of using distributed processing is huge compared to a single high-end workstation.

The architecture has been implemented on a prototypical level which allowed us to perform the evaluation. In the future we will further develop our system and perform more studies. For example, we will investigate how the presented architecture can be applied to domains other than geospatial processing. We assume by changing the DSL, the rules, and the metadata for services and data we can support other applications as well.

Acknowledgements

Research presented here is carried out within the project "IQmulus" (A High-volume Fusion and Analysis Platform for Geospatial Point Clouds, Coverages and Volumetric Data Sets) funded from the 7th Framework Programme of the European Commission, call identifier FP7-ICT-2011-8, under the grant agreement no. 318787, started in November 2012. We would like to thank Mathieu Brédif and Binh Nguyen Thai for their valuable comments and input to the evaluation section.

References

[1] Krämer M, Kehlenbach A. Interactive, GPU-Based Urban Growth Simulation for Agile Urban Policy Modelling. In: Rekdalsbakken W, Bye R, Zhang H, editors. Proceedings of the 27th European Conference on Modelling and Simulation (ECMS). Ålesund, Norway: European Council for Modelling and Simulation; 2013, p. 75–81.

[2] Cossu R, Di Giulio C, Brito F, Petcu D. Cloud computing for earth observation. In: Kyriazis D, Voulodimos A, Gogouvitis SV, Varvarigou T, editors. Data Intensive Storage Services for Cloud Environments; chap. 12. IGI Global. ISBN 9781466639348; 2013, p. 166–91. doi:10.4018/978-1-4666-3934-8.

[3] Li J, Humphrey M, Agarwal D, Jackson K, van Ingen C, Ryu Y. eScience in the cloud: A MODIS satellite data reprojection and reduction pipeline in the Windows Azure platform. In: 2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS). 2010, p. 1–10. doi:10.1109/IPDPS.2010.5470418.

[4] Giuliani G, Nativi S, Lehmann A, Ray N. {WPS} mediation: An approach to process geospatial data on different computing backends. Computers & Geosciences 2012;47(0):20 – 33. doi:http://dx.doi.org/10.1016/j.cageo.2011.10.009; towards a Geoprocessing Web.

[5] Qazi N, Smyth D, McCarthy T. Towards a GIS-Based Decision Support System on the Amazon Cloud for the Modelling of Domestic Wastewater Treatment Solutions in Wexford, Ireland. In: 2013 UKSim 15th International Conference on Computer Modelling and Simulation (UKSim). 2013, p. 236–40. doi:10.1109/UKSim.2013.62.

[6] Kuo MHA. Opportunities and challenges of cloud computing to improve health care services. J Med Internet Res 2011;13(3):e67. doi:10.2196/jmir.1867.

[7] Batty M. Big data, smart cities and city planning. Dialogues in Human Geography 2013;3(3):274–9. doi:10.1177/2043820613513390.

[8] Khan Z, Kiani SL. A cloud-based architecture for citizen services in smart cities. In: Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing. UCC '12; Washington, DC, USA: IEEE Computer Society. ISBN 978-0-7695-4862-3; 2012, p. 315–20. doi:10.1109/UCC.2012.43.

[9] Khan Z, Anjum A, Kiani SL. 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 '13; Washington, DC, USA: IEEE Computer Society. ISBN 978-0-7695-5152-4; 2013, p. 381–6. doi:10.1109/UCC.2013.77.

[10] Mell P, Grance T. The NIST Definition of Cloud Computing. Tech. Rep. 800-145; National Institute of Standards and Technology (NIST); Gaithersburg, MD; 2011.

[11] Agarwal D, Prasad S. Lessons Learnt from the Development of GIS Application on Azure Cloud Platform. In: 2012 IEEE 5th International Conference on Cloud Computing (CLOUD). 2012, p. 352–9. doi:10.1109/CLOUD.2012.140.

[12] Malawski M, Gubała T, Bubak M. Component-based approach for programming and running scientific applications on grids and clouds. International Journal of High Performance Computing Applications 2012;26(3):275–95. doi:10.1177/1094342011422924.

[13] Dean J, Ghemawat S. MapReduce: Simplified Data Processing on Large Clusters. Communications of the ACM 2008;51(1):1–13. doi:10.1145/1327452.1327492.

[14] Haller P, Odersky M. Scala Actors: Unifying thread-based and event-based programming. Theoretical Computer Science 2009;410(2-3):202–20. doi:http://dx.doi.org/10.1016/j.tcs.2008.09.019.

[15] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: cluster computing with working sets. In: Proceedings of the 2nd USENIX conference on Hot topics in cloud computing. 2010, p. 10.

[16] Eldawy A, Li Y, Mokbel MF, Janardan R. CG_Hadoop: Computational Geometry in MapReduce. In: 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL GIS 2013). 2013,.

[17] Ajiy A, Sun X, Vo H, Liu Q, Lee R, Zhang X, et al. Demonstration of Hadoop-GIS: A Spatial Data Warehousing System Over MapReduce. In: 21st ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (ACM SIGSPATIAL GIS 2013). 2013,.

[18] Xin R, Rosen J, Zaharia M, Franklin MJ, Shenker S, Stoica I. Shark: SQL and Rich Analytics at Scale. In: Proceedings of the ACM SIGMOD/PODS Conference. 2013,.

[19] Deelman E, Vahi K, Juve G, Rynge M, Callaghan S, Maechling PJ, et al. Pegasus: a workflow management system for science automation. Future Generation Computer Systems 2014;.

[20] Whitley KN. Visual programming languages and the empirical evidence for and against. Journal of Visual Languages & Computing 1997;8(1):109–42. doi:http://dx.doi.org/10.1006/jvlc.1996.0030.

[21] Rosenberg D, Scott K. Use Case Driven Object Modeling with UML: A Practical Approach. Addison-Wesley Professional; 1999. ISBN 0201432897.

[22] Belényesi M, D. Kristóf (eds.) . IQmulus public project deliverable D1.2.1 – Initial User Requirements. Tech. Rep.; 2014.

[23] Belényesi M, D. Kristóf (eds.) . IQmulus public project deliverable D1.2.2 – Consolidated User Requirements. Tech. Rep.; 2014.

[24] Belényesi M, D. Kristóf (eds.) . IQmulus public project deliverable D1.2.3 – Revised User Requirements. Tech. Rep.; 2014.

[25] Krämer M, Kießlich N, Spagnulo M. IQmulus public project deliverable D2.3.1 – Architecture Design – baseline version. Tech. Rep.; 2014.

[26] Beyer MA, Laney D. The Importance of 'Big Data': A Definition. Tech. Rep.; Gartner; 2012.

[27] Krämer M. Controlling the processing of smart city data in the cloud with domain-specific languages. In: Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing (UCC2014), the 1st International Workshop on Smart City Clouds: Technologies, Systems and Applications. 2014,To be published.

[28] Khan ZA, Ludlow D, McClatchey R, Anjum A. An architecture for integrated intelligence in urban management using cloud computing. CoRR 2012;abs/1202.5483.

[29] Zdun U, Capilla R, Tran H, Zimmermann O. Sustainable architectural design decisions. IEEE Software 2013;30(6):46–53. doi:10.1109/MS.2013.97.

[30] Breivold HP, Crnkovic I, Larsson M. A systematic review of software architecture evolution research. Information and Software Technology 2012;54(1):16–40.

[31] Wong WE, Christensen HB, Hansen KM. An empirical investigation of architectural prototyping. Journal of Systems and Software 2010;83(1):133–42.

[32] Bosch J. Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach. Addison-Wesley Professional; 2000. ISBN 0201674947.

[33] Olston C, Reed B, Srivastava U, Kumar R, Tomkins A. Pig Latin: A Not-so-foreign Language for Data Processing. In: Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. SIGMOD '08; New York, NY, USA: ACM. ISBN 978-1-60558-102-6; 2008, p. 1099–110. doi:10.1145/1376616.1376726.

[34] A. Eldawy and M. Mokbel . Pigeon: A Spatial MapReduce Language. In: Proceedings of the IEEE International Conference on Data Engineering (ICDE). 2014,.

[35] Depardon B, Mahec GL, Séguin C. Analysis of six distributed file systems. Tech. Rep.; HAL; 2014.

[36] Zaharia M, Chowdhury M, Franklin MJ, Shenker S, Stoica I. Spark: Cluster Computing with Working Sets. In: Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing. Berkeley, CA, USA: USENIX Association; 2010,.

[37] Fielding RT. Architectural styles and the design of network-based software architectures. Ph.D. thesis; University of California; 2000. doi:10.1.1.91.2433.

[38] Gaševic D, Djuric D, Devedžic V. Model Driven Engineering and Ontology Development. 2 ed.; Springer; 2009. ISBN 978-3-642-00282-3.

[39] Provos N, Mazires D. A future-adaptable password scheme. In: USENIX Annual Technical Conference, FREENIX Track. 1999, p. 81–91.

[40] Malawski M, Meizner J, Bubak M, Gepner P. Component approach to computational applications on clouds. Procedia Computer Science 2011;4(0):432 –41. doi:http://dx.doi.org/10.1016/j.procs.2011.04.045; proceedings of the International Conference on Computational Science, {ICCS} 2011.