EXCLAIM framework: a monitoring and analysis framework to support self-governance in Cloud Application Platforms

A thesis submitted to The University of Sheffield for the degree of Doctor of Philosophy

Rustem Dautov

Department of Computer Science / South-East European Research Centre

June 2016

Acknowledgements

My warmest words of gratitude go to my academic supervisors – Iraklis Paraskakis and Mike Stannett – without whom this work would never have been completed.

I would also like to thank South-East European Research Centre and CITY College Computer Science Department for offering me the opportunity to pursue the PhD degree, and all the nice people working there for their warm support during my PhD studies.

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7-PEOPLE-2010-ITN) under grant agreement N°264840.

Abstract

The Platform-as-a-Service segment of Cloud Computing has been steadily growing over the past several years, with more and more software developers opting for cloud platforms as convenient ecosystems for developing, deploying, testing and maintaining their software. Such cloud platforms also play an important role in delivering an easily-accessible Internet of Services. They provide rich support for software development, and, following the principles of Service-Oriented Computing, offer their subscribers a wide selection of pre-existing, reliable and reusable basic services, available through a common platform marketplace and ready to be seamlessly integrated into users' applications. Such cloud ecosystems are becoming increasingly dynamic and complex, and one of the major challenges faced by cloud providers is to develop appropriate scalable and extensible mechanisms for governance and control based on run-time monitoring and analysis of (extreme amounts of) raw heterogeneous data.

In this thesis we address this important research question – **how can we support self-governance in cloud platforms delivering the Internet of Services in the presence of large amounts of heterogeneous and rapidly changing data?** To address this research question and demonstrate our approach, we have created the Extensible Cloud Monitoring and Analysis (EXCLAIM) framework for service-based cloud platforms. The main idea underpinning our approach is to encode monitored heterogeneous data using Semantic Web languages, which then enables us to integrate these semantically enriched observation streams with static ontological knowledge and to apply intelligent reasoning. This has allowed us to create an extensible, modular, and declaratively defined architecture for performing run-time data monitoring and analysis with a view to detecting critical situations within cloud platforms.

By addressing the main research question, our approach contributes to the domain of Cloud Computing, and in particular to the area of autonomic and self-managing capabilities of service-based cloud platforms. Our main contributions include the approach itself, which allows monitoring and analysing heterogeneous data in an extensible and scalable manner, the prototype of the EXCLAIM framework, and the Cloud Sensor Ontology. Our research also contributes to the state of the art in Software Engineering by demonstrating how existing techniques from several fields (i.e., Autonomic Computing, Service-Oriented Computing, Stream Processing, Semantic Sensor Web, and Big Data) can be combined in a novel way to create an extensible, scalable, modular, and declaratively defined monitoring and analysis solution.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

During the last decade, cloud computing has passed a long way from a term mainly known to Information Technology (IT) professionals and computer scientists to a buzzword, widely-known and recognisable by ordinary users. Indeed, cloud technologies, to a greater or lesser extent, are nowadays involved in almost every area of our daily life and economy – ubiquitous cloud services are rapidly transforming the way we do business, maintain our health, educate and entertain ourselves (Dautov et al., 2014*b*). Ever-increasing numbers of ordinary users are using mobile cloud apps and storing their data in cloud-based storage, while industries are anticipating more and more economic benefits from migrating and running IT tasks in cloud environments. This paradigm shift from traditional, in-house computation and storage to remote virtualised cloud ecosystems would not be possible without the recent advances in computing, networking, software, hardware and mobile technologies. Technological achievements in these domains go hand in hand with cloud computing to support and facilitate its increasing popularity and adoption.

In particular, the Platform-as-a-Service (PaaS) segment of cloud computing has been exhibiting steady growth over the past several years, with an ever-increasing number of software developers opting for cloud platforms as convenient ecosystems for their applications throughout the whole software lifecycle. IDC,[1] a major global research and advisory company, reported 46.2% growth in the public PaaS market in 2012 and predicts that by 2017 it will keep increasing by 30% annually, to reach a turnover of $14 billion (Mahowald et al., 2013). Such rapid growth and the success of the PaaS segment can be explained by the promising business opportunities offered to enterprises. With this cloud computing delivery model, the target audience – typically, software developers – is offered an appealing en-

[1] `http://www.idc.com/`

vironment and tooling support for developing, testing, deploying, and maintaining their software on a remote virtualised platform, paying for computational resources only when they are really needed, without the requirement to own or manage the underlying hardware infrastructure and Operating System (OS). In these circumstances, a platform offered as a service typically consists of a Virtual Machine (VM) with an OS, execution environment, simple data storage, and development tools, running on a shared multi-tenant physical server. Following the principles of Service-Oriented Computing (SOC) (Wei and Blake, 2010), a subset of such cloud platforms also offer their subscribers a selection of pre-existing and reusable services, which can be seamlessly integrated into users' applications and thus allow developers to concentrate on their immediate business tasks and exempt them from 're-inventing the wheel'. Such basic services, for example, may include *data storage*, *queue messaging*, *searching*, *E-mail and SMS*, *logging* and *caching*, among others.[1] The emergence of SOC in the early 2000s (Curbera et al., 2002, Huhns and Singh, 2005, Papazoglou et al., 2003, Ross and Westerman, 2004, Roy and Ramanujan, 2001) opened new business opportunities for enterprises who migrated their IT systems from traditional monolithic approaches towards highly modular and re-usable service-based architectures. This allowed organisations to develop distributed software systems in a short period of time by dynamically assembling basic services supplied by multiple service providers and hosted on different hosting platforms (Wei and Blake, 2010). As a result, instead of spending their effort and time on indirect routine tasks, such as re-implementing already existing components (e.g., an authentication mechanism for a web site), or migrating and configuring a legacy database, with just a few mouse clicks developers have the capacity to easily integrate an existing service into their applications, thereby accessing a tested, optimised and reliable solution from an experienced vendor, typically through a transparent and easy-to-understand pricing scheme. Usually, cloud platforms offer such basic add-on services through a central marketplace, where platform and third-party services can be accessed. Accordingly, cloud platform customers only have to choose a convenient option from an (ever-growing) selection of existing authentication and data storage solutions.[2] Cloud platforms providing basic services discoverable and accessible via a service marketplace are nowadays seen as a one of the main contributing factors to delivering reliable Internet of Services (IoS). Given this essential and distinctive from other PaaS solutions characteristic, which allows software developers to concentrate on their immediate business-oriented tasks, we refer to this subset of PaaS offerings

[1] Heroku, for example, provides over 150 add-ons (`http://addons.heroku.com/`).

[2] In this case, customers will only have to move their data to the cloud, which is still much simpler than building up a whole database server in the cloud from scratch.

as Cloud Application Platforms (CAPs) to highlight their application-centric and business-oriented nature.

1.1 Motivation: Lack of self-governance in Cloud Application Platforms

Attractive from the business perspective, the model offered by CAPs, however, is not a 'silver bullet', whether for enterprises willing to see their software up and running in the clouds, or CAP providers seeking to offer better service and thus attract more customers. With the increasing flexibility and reduced time-to-market, come emerging challenges as to how the available cyberspace resources and services should be properly managed (Dautov et al., 2014*b*). This flexible model for application development, in which software systems are assembled from existing components in a manner similar to a Lego® construction set, generates complex interrelationships between services and user applications, and eventually leads to a situation where CAP providers have to cope with the ever-growing complexity of entangled cloud environments and face new challenges as to how such systems should be governed, maintained and secured.

Platform (self-)governance is becoming of utmost importance, as cloud platforms compete, striving to deliver an even wider selection of services and accommodate even more user applications. It follows that in order to maintain the stable and balanced performance of the CAP and deployed applications, providers should be able to exercise control over all potentially critical activities associated with i) the run-time execution of services and applications, ii) the introduction of new services and applications, and iii) the modification of existing ones (Kourtesis et al., 2012). However, with the on-going paradigm shift towards cloud computing in general and its PaaS segment in particular, the complexity of tomorrow's service-based cloud environments is expected to outgrow our capacity to manage them in a manual manner (Brazier et al., 2009, Dautov et al., 2013). Today's CAPs (e.g., Google App Engine,[1] Windows Azure,[2] and Heroku[3]) already host numerous services, ranging from simple operations to complicated business logic, all ready to be integrated into an ever-expanding range of service compositions. However, they do not offer support for user-customised monitoring and problem detection to facilitate automated governance and management of the resulting service-based systems as part of their offering (Wei and Blake, 2010). In the era

[1] `http://appengine.google.com/`
[2] `http://www.windowsazure.com/`
[3] `http://www.heroku.com/`

of the IoS, when applications are increasingly dependent on third-party solutions offered as services, a failure of an external component at one point may lead to malfunctioning of a whole cloud-based application system, without the hosting platform noticing it. Presently, CAPs are incapable of detecting and reacting to such situations (Dautov et al., 2012) – e.g., they cannot understand whether a response from a service is correct or not, substitute a malfunctioning or underperforming service with another, or balance resource consumption accordingly.

The existing limitations make it necessary for software engineers to remain involved in the application life cycle after it has been deployed to the cloud environment. That is, they have to monitor application at run-time, detect potential critical situations, and apply adaptations, which need to be performed manually by rewriting the application/platform programming code, recompiling, redeploying the application or restarting the platform. To some extent, this challenge is analogous to the problem faced in the 1920s, when increased telephone usage led to the introduction of the automatic branch exchanges so as a substitute for the insufficient manual labour of human operators (Huebscher and McCann, 2008). As we explore in more detail below, a similar goal is pursued by the Autonomic Computing research, which aims at minimising human involvement in complex computing systems. By bringing together many fields of IT (e.g., Control Theory, Agent Theory, Software Engineering, Information and Data Management, etc.) (Huebscher and McCann, 2008), Autonomic Computing serves to create systems that are capable of self-management (that is, autonomic systems) – a key feature of complex and dynamic computing systems, including cloud ecosystems.

The sample scenario described above, albeit it already represents a specific and pressing gap to be addressed by Heroku, is intended to highlight and bring to the reader's attention the general systematic lack of scalable control and governance capabilities within CAPs. Providing customers with more visibility into service resource consumption is important, but reflects only a single dimension of the envisaged governance and control mechanisms in CAPs. There can be identified three key stake-holders involved in the CAP governance process:

- The **CAP provider**, who is interested in executing constant control over platform resources provisioned to consumers. In particular, the provider needs to ensure that the amount of virtualised abstracted resources provisioned to add-on services and user applications never exceeds the actual physical amount of underlying hardware infrastructure.

- The **CAP consumer**, who wants to be aware of the current amount of resources utilised by his software. The consumer also needs to be notified

whenever his resource consumption is approaching critical levels.

- The **third-party service provider**, who offers his add-on services through the platform marketplace. His role in the governance process is to notify the CAP providers and consumers of how his respective services have to be monitored and treated.

There are two main aspects which have to be taken into consideration. Firstly, CAPs are complex systems, which span across all three levels of cloud computing (i.e., Infrastructure-as-a-Service (IaaS), PaaS, and Software-as-a-Service (SaaS)), and, therefore, governance activities are not only limited to services and resources at the platform level. For example, they may also include the monitoring of infrastructure resources or performing higher-level business analytics to support Business Process Management (BPM). Secondly, CAPs are highly dynamic systems and are expected to change on a daily basis (e.g., new user applications deployed, new services added). Given this, the challenge of the CAP governance is also associated with creating an extensible and scalable architecture, which would allow adapting to emerging requirements in a seamless and transparent manner. In this context, *extensibility* can be defined as the ability of a system to be extended with new functionality (or to modify existing functionality) with minimum effect on the system's internal structure and data flow. Typically, extensibility entails that recompiling or changing the original source code is unnecessary when changing the system's behavior. An extensible architecture is usually achieved by applying modularity principles – that is, by breaking down a 'monolithic' application into multiple loosely-coupled components – and using declarative programming languages, which allow for seamless modification of indivudual components without interfering with the rest of the system. As for *scalability*, in the context of the present research it is defined as the ability of a system to handle a growing amount of work, or its potential to accommodate that growth, while maintaining a stable level of service (Bondi, 2000).

Accordingly, effectively managing CAPs inevitably means that resources will need to become increasingly autonomous and capable of managing themselves with minimum human involvement – that is, self-managing (Dautov et al., 2014*b*). The autonomic behaviour of a computer system is typically achieved through implementing closed control loops whose main principle is to iteratively observe the system's surrounding environment and context, and then react accordingly if needed, without manual intervention. One of the main reference models for implementing such closed adaptation loops is the Monitor-Analyse-Plan-Execute-

Knowledge (MAPE-K) model from IBM[1] (Kephart et al., 2007). It consists of four steps – Monitor, Analyse, Plan, Execute – and a unifying Knowledge component which conceptually represents all the information needed to perform these four activities.

In this regard, data monitoring and timely problem detection in the context of complex service-based environments can be seen as a multi-faceted challenge, which involves the following key aspects:

- Volume: potentially extreme scale of data created, processed and stored
- Velocity: on-the-fly processing of streaming data
- Variety: data is generated by multiple heterogeneous sources, and comes in many different forms and formats
- Veracity: data is often uncertain, flawed, or rapidly changing

Enabling CAPs with autonomic behaviour means constantly monitoring the current state of the CAP ecosystem, as the monitoring process can be seen as an entry point for the whole adaptation process – observations obtained at this stage act as a trigger for potential adaptations; at a later stage they are fed to the analysis and planning components, which, in their turn, diagnose potential problems and suggest relevant adaptations to be applied to the system. Such monitoring and analysis activities, for example, include measuring resources currently available to customers, identifying bottlenecks and breakdowns, checking for Service Level Agreement (SLA) violations, etc. Consequently, if a service fails to meet expected requirements, the CAP provider needs to detect and diagnose this situation as quickly as possible, and to take necessary steps to provide seamless hand-overs from one service to another.

1.2 Towards the solution: the EXCLAIM framework

Deployed applications and dependant services (which are monitored subjects in this context, and whose operational behaviour has to be observed and interpreted) can be seen as self-contained entities, which form complex interrelationships and constantly generate raw data to be monitored and analysed by the CAP providers.

[1] Another example of a reference model for creating closed adaptation loops is Collect – Analyze – Decide – Act (CADA) (Dobson et al., 2006), which is similar to MAPE-K in the general principle of collecting, interpreting and acting upon data. Unlike CADA, however, the MAPE-K model explicitly has Knowledge as its key component. As explained in Chapters 2.2 and 5, the knowledge base is of paramount importance in the context of this research work, and, therefore, MAPE-K was chosen as our main point of reference.

Indeed, with the ever-increasing number of offered services and deployed applications,[1] CAPs are coming to resemble complex sensor network graphs where nodes are deployed applications and services, and edges are the dependencies between them. This allows us to draw parallels between CAPs and sensor networks. Looking at CAPs from this perspective allows building on the work carried out by the Sensor Web research community in the context of dynamic monitoring and analysis of continuously flowing streams of data. In particular, their Sensor Web Enablement (SWE) initiative (Botts et al., 2008) focuses on enabling an interoperable usage of discoverable and accessible sensor resources. In other words, it aims at enabling timely integration of separate heterogeneous sensor networks into the single global information infrastructure – a very similar goal to our own, which we aim to address by creating an Extensible Cloud Monitoring and Analysis (EXCLAIM) framework for service-based cloud environments.

While challenges associated with timely processing of sensor data have been relatively successfully tackled by the advances in networking and hardware technologies (Liang et al., 2005, Akyildiz et al., 2002*b*,*a*), the challenge of properly handling data representation and semantics of sensor descriptions and sensor observations is still pressing. In the presence of multiple organisations for standardisation, as well as various sensor hardware and software vendors, overcoming the resulting heterogeneity remains one of the major concerns for the SWE initiative (Botts et al., 2008). An analogous problem exists in CAPs, where deployed applications and services, provided by different independent software vendors, are characterised with high degrees of diversity and heterogeneity.

This lack of a unified data representation has been addressed in the context the Semantic Sensor Web (SSW) – a promising combination of two research areas, the Semantic Web and the Sensor Web (Sheth et al., 2008). Using the Semantic Web technology stack to represent data in a uniform and homogeneous manner, it provides enhanced meaning for sensor descriptions and observations so as to facilitate situation awareness along with support for timely, dynamic stream processing (Dautov et al., 2014*a*). Given our interpretation of CAPs as distributed networks of logical sensors and applying the SSW approach to this domain, we aim to represent highly-heterogeneous data of CAPs in a uniform manner, and, at the same time, facilitate interoperability, human-readability and, most importantly, formal analysis of the semantically-annotated data streams.

Another important aspect of this Sensor Web-inspired approach is the opportunity to address the Big Data challenge associated with processing vast amounts of raw data within CAPs. Sensor networks are characterised by extreme amounts

[1] Heroku, for example, reports over 1.5 million deployed applications (Harris, 2012).

of streaming data, and existing solutions aim at processing avalanches of sensor readings with minimum delay in a dynamic and scalable manner. The SSW community has addressed the issue of prompt stream processing by introducing so-called continuous query languages for semantically-annotated data streams. These query engines extend the functionality of the query language for static semantic datasets with support for temporal and sequential operators – that is, support for streaming, real-time data. As a result, massive amounts of constantly updated sensor information are not permanently stored, but rather processed in a dynamic, in-memory fashion so as to achieve faster execution and eliminate the unnecessary burden of storing this big data. It is worth mentioning, however, that existing stream processing solutions, as yet immature and not well optimised, often tend to suffer from the scalability issue, and therefore can be even further enhanced by applying two fundamental principles of Big Data processing – data partitioning and parallel processing (Dean and Ghemawat, 2008, Chaiken et al., 2008) – which enable processing sub-streams of sensor observations in parallel, and thus achieve faster execution.[1]

Following the MAPE-K model for implementing closed adaptation loops and treating CAPs as networks of logical software sensors, we created a proof-of-concept prototype and validated it with a Heroku case study. When implementing the prototype, we re-used existing techniques from the SSW domain:

- Web Ontology Language (OWL) to develop a common architectural model of the managed CAP environment;
- Resource Description Framework (RDF) to represent semantic data streams;
- RDF stream processing techniques to query the streaming data;
- Formal reasoning capabilities of OWL and Semantic Web Rule Language (SWRL) to analyse and detect potentially critical situations.

Case study results show that we are able to successfully detect critical situations within Heroku when running relatively small-scale experiments. However, the situation gets worse when the workload (i.e., amount of streamed data to be analysed by the framework) increases, as the framework faces the scalability issue, associated with i) a single-threaded, pipelined approach to data stream processing, and ii) low scalability of formal reasoning in general. To address this challenge as well, we have deployed the framework on top of IBM InfoSphere Streams – a part of

[1]Each of the background theories and related technologies, relevant to this research work, are described in details in corresponding chapters of this thesis.

IBM's solution stack for Big Data processing, which focuses specifically on dynamic streaming data. This deployment allowed us to split the main data stream with monitored values into several sub-streams and process them in a parallel and scalable manner. Experimental results also indicate noticeable improvements in processing time with the IBM Streams deployment.

1.3 Aim of the thesis

With the presented thesis, we aim to address the outlined challenges by raising and answering the main research question – **how can we support self-governance in cloud platforms delivering the Internet of Services in the presence of large amounts of heterogeneous and rapidly changing data?** effectively, answering this main question requires addressing several sub-questions:

- **What are the key challenges to be addressed?** Answering this question requires through classification of the problem domain of service-based cloud platforms, and identification of the main existing challenges and requirements for the future solution. This has to be done in a comprehensive manner from several perspectives – that is, it is important to identify key stakeholders and their respective roles in the process of platform governance. As a result, addressing these issues is expected to outline techniques, which can be potentially re-used and applied.

- **How should cloud software and add-on services be treated to enable data collection and analysis?** It is important to understand what kind of data is expected to be monitored, whether it can be collected in a non-intrusive manner, and, if not, how to minimise potential intrusions to application source code.

- **How to engineer the resulting software framework, addressing identified challenges?** This refers to designing and implementing the framework following the established software engineering practices, including support for scalability, extensibility, and performance. This also includes addressing the volume, velocity, variety and veracity of data generated and collected within CAPs.

- **How to model the internal architecture of the monitored cloud environment?** This involves understanding requirements for a suitable modeling language, finding the right language, and, finally, thoroughly and precisely modeling the cloud platform ecosystem.

- **How to enable a fine-grained, differentiated, and user-customised approach to data collection and monitoring?** This requires designing and implementing the monitoring in such a manner, that the involved parties (i.e., the CAP provider, the CAP consumer, and the third-party add-on provider) are given an opportunity to participate in the process of cloud platform governance.

By addressing these research questions, the presented work contributes to the domain of Cloud Computing, and in particular to the area of autonomic and self-managing capabilities of service-based cloud platforms. It also contributes to the state of the art in Software Engineering by demonstrating how existing techniques from several fields (i.e., Autonomic Computing, Stream Processing, SSW, and Big Data processing) can be combined in a novel way to create an extensible, scalable, modular, and declaratively defined monitoring and analysis framework.

1.4 Contributions of the thesis

1. **Novel concept of software self-containment**: this concept relies on interpreting software elements as logical sensors, and cloud platforms as distributed networks of such sensors, and allows for individual software sensors to be equipped with respective self-governance knowledge (e.g., self-diagnosis and self-adaptation policies) to enable decoupled, modular and distributed organisation of the knowledge base. As opposed to computationally expensive reasoning over a potentially heavy-weight, 'monolithic' knowledge base, with such organisation, it is possible to limit analysis activities to a specific set of policies related to a particular scenario, and thus minimise the amount of unnecessary computations. This contribution is explained in Section 5.3.1.

2. **The overall approach to data monitoring and analysis to support self-governance in service-based cloud platforms**: the approach involved a considerable amount of preliminary research work – that is, it relies on thorough analysis of the problem, classification of existing challenges, and identification of suitable techniques to be utilised in this respect. The approach itself demonstrates capabilities for declarative definition of knowledge, extensibility, modularity, and scalability. Based on the concept of interpreting software services as sensor networks, we devised a new approach to implement monitoring and analysis capabilities. The proposed approach is a combination of several technologies and research domains – namely, Cloud Computing, SSW, Stream Processing, Big Data Processing, and Software Engineering –

and potentially can be re-used and applied to other problem domains related to processing large amounts of heterogeneous data. Please refer to Chapter 5 for further details.

3. **Conceptual design of the EXCLAIM framework**: the conceptual design of the proposed EXCLAIM framework may serve as a reference model to implement similar frameworks using other components and technologies. As explained in further detail below, certain components of the EXCLAIM implementation could be seamlessly replaced by other existing alternatives, whereas the overall conceptual architecture of the framework would remain the same. The conceptual design is explained in Section 5.2.

4. **Two-tier Cloud Sensor Ontology**: the upper tier of this ontology models a cloud platform environment in a generic manner, and therefore can be extended appropriately to model a particular cloud platform. The lower level models the specifics of Heroku and can also be potentially re-used or extended in various applications dealing with this CAP. More details on the ontology can be found in Section 6.3.

5. **Novel concepts of service criticality and application profiling**: these concepts allow for a more fine-grained approach to monitoring and analysing individual software elements within cloud platforms, and also enable more optimised utilisation of available resources by triggering analysis activities only when they are really required. Please refer to Section 5.3.2 for further details.

1.5 Thesis outline

Chapter 2 – Background theory: Cloud Computing and Autonomic Computing is two-fold. First, it explains the main context of the presented research effort, namely – CAPs, which are cloud platforms characterised with an extensive support for software development by offering a rich selection of reusable basic services. In the presence of large amounts of heterogeneous data collected from services and applications, CAPs require increased capabilities for self-governance. Accordingly, the second part of the chapter explains the main principles of autonomic computing, which serves to underpin our proposed approach to support self-governance in service-based cloud platforms. The chapter briefs the reader on the fundamentals of autonomic computing, and summarises levels of self-management in computing

systems. The chapter also introduces the MAPE-K reference model for creating autonomic systems, which will act as the main reference model to implement our proposed framework.

Chapter 3 – Related technologies: Big Data processing and the Semantic Sensor Web introduces the notion of Big Data, which poses novel research challenges associated with how ever-increasing volumes of constantly generated heterogeneous data have to be efficiently handled. The chapter summarises the main challenges associated with Big Data processing, known as the 'four Vs' of Big Data – Volume, Velocity, Variety and Veracity, and introduces the concept of Stream Processing for performing run-time analysis over streaming data. Next, the chapter proceeds with the Semantic Sensor Web, which is a combination of the Semantic Web and the Sensor Web research areas, serving to overcome data and format heterogeneity currently present in sensor networks. The chapter concludes with an overview of existing technologies – namely, the Semantic Sensor Network (SSN) ontology and corresponding stream processing engines – which are successfully utilised by the community, and which have been used in our own research as well.

Chapter 4 – State of the art in cloud platform self-governance provides a comprehensive view on the current state of the art in the domain of monitoring and analysis of service-based cloud platforms. To better understand existing challenges and state of the art in self-governance at the PaaS level, the chapter first provides an overview of self-managing capabilities at the IaaS and SaaS levels of cloud computing. It then summarises the existing body of relevant research efforts and clusters them into five main groups. This clustering is intended to provide a better understanding of existing solutions and approaches. By clustering relevant approaches we also conducted a critical analysis of the state of the art, distilled several observations and identified research gaps to be addressed by our own research work.

Chapter 5 – Conceptual architecture of the EXCLAIM framework outlines main characteristics of data monitoring and analysis within service-based cloud platforms from the Information Management point of view, and introduces a novel concept of treating CAPs as software sensor networks. Based on this novel interpretation, it then presents a conceptual design of our EXCLAIM framework, which consists of three main elements. The chapter also explains additional enhancements to the main design, which serve to improve the framework in terms of its modularity and capabilities for declarative definition of the knowledge base.

Chapter 6 – Implementation details of the EXCLAIM framework provides lower-level implementation details of the framework architecture. First, it explains the three main components of the framework responsible for data triplification, querying and reasoning. Then, the chapter proceeds with an explanation of a two-tier ontology and a set of linked policy extensions, which constitute the knowledge base of the framework. The chapter also briefs the reader on the actions, which need to be taken in order to deploy the framework on a different CAP and start using it.

Chapter 7 – Proof of concept: monitoring and analysis of Heroku add-on services with the EXCLAIM framework combines the material from the two previous chapters and demonstrates how the framework functions based on a case study. The case study is intended to demonstrate the framework's performance, scalability and extensibility (i.e., declarative definition of the knowledge base and modularity). The chapter measures the performance of the framework under various workloads. Finally, it also describes how the scalability issue was addressed using an existing Big data processing solution, and presents updated experimental results.

Chapter 8 – Evaluation and discussion first evaluates the presented approach with respect to its performance, scalability and extensibility. Then, it also summarises and discusses potential benefits and shortcomings associated with our approach.

Chapter 9 – Conclusion summarises the whole thesis with an overview of main research contributions and answers to the research questions raised in the introduction. The chapter also outlines several directions for further work, and concluder with the author's view on the presented work - the latter contains personal evaluation and impression from the conducted research.

Part I

Background theory and related technologies

In order for the reader to understand the value and contribution of the presented research work, we first start with an in-depth explanation of the context of the research work (see Chapter 2). In our work, we are focussing on a subset of cloud platforms, which are characterised with rich support for rapid application development. Such platforms, which we call Cloud Application Platforms, host a service marketplace – a collection of generic reusable add-on services, which can be easily accessed and integrated into customers' cloud-hosted software systems. By hereby contributing to reliable and easily-accessible Internet of Services, such platforms are, however, becoming increasingly complex, tangled and unstable, and calling for novel mechanisms for self-governance and control. This, accordingly, became the main motivation behind the presented research work – as it will be explained in more details below, service-based cloud platforms have to be equipped with mechanisms for self-governance, which would rely on an extensible and scalable capabilities for data monitoring and analysis.

Notions of self-governance and self-management stem from the research area of Autonomic Computing (see Section 2.2), which aims at bringing involvement of human operators in complex computer system to its minimum. In order to support self-management, an autonomic system relies on a constant process of assessing the surrounding context, interpreting monitored information and acting upon these observations if needed. Similarly, in order to support self-governance in cloud platforms, our approach implements the MAPE-K reference model as its fundamental underpinning. Conceptually, the whole self-management cycle can be split into four interconnected steps – namely, Monitoring, Analysis, Planning and Execution. These four activities share a common Knowledge base, which is the central component of the self-adaptation model, and, consequently, of our approach as well.

Aiming at developing a monitoring and analysis framework for such complex and dynamic environments as cloud platforms, we cannot avoid facing the challenge of processing large amounts of rapidly changing heterogeneous data – or, simply put, the Big Data challenge. Accordingly, in Section 3.1, we provide an overview of this relatively novel research area, paying special attention to the state of the art in processing streaming data. Stream processing differs from the traditional static approaches in that it handles data in memory, without permanently storing it on hard disk. In the context of the presented research work, stream processing capabilities are expected to help us in achieving timely and prompt detection of potential problems.

However, the streaming approach on its own is not sufficient to effectively analyse highly-heterogeneous data, present in service-based cloud environments. Variety of monitored subjects (i.e., add-on services, deployed applications, platform components) dictates the need for a uniform and semantically-enriched data representation of collected values. Stream processing over semantically-annotated data has been the focus of the Semantic Sensor Web research, whose main goal is to overcome existing heterogeneity in sensor networks by introducing a common semantic vocabulary of terms and developing appropriate technologies for processing semantic data streams. To a certain extent, the experience and techniques from this novel research area can be applied to other stream processing domains, where key challenge is to overcome data heterogeneity with a common semantic vocabulary. In Section 3.2, we provide a detailed overview of these concepts, and explain how we can benefit from the Semantic Sensor Web research.

After reading Part I of the document, the reader is expected to be familiar with all necessary information to proceed to Part II, which will first provide an overview of the state of the art in the considered research direction, and then proceed with the actual description of our approach and the EXCLAIM framework.

Chapter 2

Background theory: Cloud Computing and Autonomic Computing

This chapter starts with explaining concepts of services, SOC and IoS, which underpin the motivation and rationale behind transforming the way modern IT is provisioned into a model, where users can access computing resources remotely in a transparent, flexible, pay-per-use manner. One of the main examples, demonstrating benefits of migrating from traditional, in-premises software to remote, virtualised, service-based model are, undoubtedly, clouds. The chapter introduces the notion of Cloud Computing, as well as existing classifications of delivery and deployment models. Special attention is paid to the PaaS segment, and, more specifically, to CAPs – a subclass of PaaS offerings, which follow the SOC principles by not just providing cloud computing utilities as a service, but additionally offer software developers a range of already existing, re-usable and remotely accessible basic services. By listing and explaining main benefits of this attractive cloud model, we also bring to the reader's attention the existing challenge of insufficient capabilities for self-governance, caused by the ever-growing complexity and 'dynamicity' of CAPs.

This challenge can be addressed by applying principles of Autonomic Computing, and the second part of the chapter familiarises the reader with this research area, which sets its goal to decrease the role of human administrators in run-time operation of complex computing systems. The chapter briefly introduces the history of autonomic computing, and lists main characteristics an autonomic system typically demonstrates. Also, as explained below, we employ the fundamental reference model for implementing autonomic systems, known as MAPE-K, as a

baseline for our work. Understanding this model in detail is important to follow the further explanation of the EXCLAIM framework, which is a partial implementation of the MAPE-K loop. Also, to position our work within the scope of the existing technological and research efforts aiming at creating autonomic systems, we present and explain 5 levels of autonomic behaviour of computing systems. The chapter concludes with an overview of autonomic features currently present in cloud computing in general and CAPs in particular – this is expected to demonstrate the need for more intensive research efforts to be put into exploring autonomic capabilities at the PaaS level.

2.1 Context of the research work: Internet of Services, Cloud Computing and Cloud Application Platforms

In the last 30 years software development experienced several paradigm shifts – after Object-Oriented Computing (OOC) in the 80s (Rumbaugh et al., 1991) and component-based software engineering in the 90s (Heineman and Council, 2001), we are now witnessing how the Service-Oriented Architectures (SOAs) are transforming modern IT. It is an evolutionary approach to building distributed software systems in a loosely-coupled and technology- agnostic manner. It builds upon and further extends the principles, successfully introduced and adopted in OOC and component-based software development, such as self-description, explicit encapsulation, and run-time functionality loading (Microsoft Corporation, 2015).

The fundamental building block of SOA is a service. Broadly speaking, services can be defined as reusable software components, remotely accessible in a loosely-coupled and technology-agnostic manner via a standard well-defined interface (Wei and Blake, 2010). They are designed to perform simple, granular functions with limited knowledge of how other components of a larger SOA system are implemented and communicate. Accordingly, SOA is a design pattern based on a simple, yet efficient principle – applications provide distinct pieces of their software functionality as services to other applications via an established and standardised protocol. The main idea of SOA is to rapidly assemble distributed software systems from existing services in such a way that they can be easily modified if needed. By abstracting the implementation details and only exposing their Application Programming Interfaces (APIs), various heterogeneous services can be integrated into Service-based Application (SBA) systems, which do not depend on the underlying implementation – they are independent of any vendor, product or technology.

Correct implementation of the SOA principles leads to a whole new paradigm

in computing – SOC. In SOC, software systems are organised and operate in such a way that various pieces of distributed computing capabilities may be under control of different ownership domains (MacKenzie et al., 2006). One of the biggest benefits of such an organisation is its agility – business processes implemented in a service-based, loosely-coupled fashion are much easier to change and evolve compared with 'monolithic' applications, constrained with underlying technologies, which require much more time to be adjusted if/when needed (Microsoft Corporation, 2015). Furthermore, this modular, service-based approach to implementing software systems paved the way for applying a similar approach to breaking down heavy-weight 'monolithic' applications within a single enterprise. This architecture, which has become known as *microservice architecture* (Newman, 2015), implies that enterprise-level software has to be designed and implemented as a suite of small independent services communicating through lightweight mechanisms – typically with HyperText Transfer Protocol (HTTP) calls (Lewis and Fowler, 2015). The microservice architecture differs from the traditional component-based approach to software engineering, where software libraries are imported into the application, in that it is more loosely-coupled and technology-agnostic. Internal implementation of individual microservices is hidden, and the communication between the services takes place by means of lightweight network communication and standard and stable APIs. Another difference of this novel approach is that microservices are implemented around business capabilities, as opposed to conventional way of breaking down applications into functional (i.e., technological) layers – data storage, server-side logic, user interface, etc. – and, accordingly, employing dedicated development teams. In the latter case, a change to, for example, the database layer may affect other layers and requires participation of all relevant teams so as to implement required changes. In contrast, a cross-functional microservice is a broad-stack implementation, which involves various engineering skills. In these circumstances, changes to microservices are made locally by a team responsible for a particular microservice, and do not affect the rest of the enterprise's development staff.

These promising opportunities of SOC are becoming even more attractive today, when many organisations are trying to cope with rapidly changing market requirements, such as evolving customer needs and new business processes with minimum expenditures (Wei and Blake, 2010). Consequently, competitiveness requires that companies continually modify their IT systems by creating new systems and retiring old solutions in a relatively short period of time. These requirements have been successfully addressed by SOC – the loosely-coupled nature of the underlying IT resulted in loosely-coupled and easily modifiable business

processes. SOC allowed enterprises to reduce the need to develop new software components each time a new business process arises, but rather use services as basic blocks to construct rapid, low-cost, yet secure and reliable applications (Wei and Blake, 2010). It is important to understand that services are designed and implemented to remain static and stable, whereas the configuration of SBAs – that is, the way services are connected and interact with each other – is supposed to change and evolve. It means, that in an occasion when emerging business needs dictate new requirements to the supporting enterprise IT systems, it should not lead to serious, effort-intensive manual changes to the software source code. For example, let us consider a case when an SBA relying on a third-party e-mail messaging service, needs to perform an additional anti-spam check whenever an email is sent/received. In these circumstances, the SOC principles suggest that either a) an additional third-party service, responsible for anti-malware inspections, should be integrated into the system and all the email traffic should be routed through it,[1] or b) the enterprise should simply replace the existing e-mail service with a more advanced alternative, which has built-in anti-spam support.[2] In both cases, no massive source code modifications are required. In this sense, services can be seen as 'black boxes' which only expose to the outer world their description and interfaces so as to enable users to discover them and access their functionality respectively. They are stable and rather static entities, and in order to achieve this, services rely on well-defined, standardised and technology-independent interfaces and self-description formalisms, which allow configuring them to be integrated into SBAs.

The ever-increasing conversion of the modern IT into service orientation, as well as the emergence of the so-called Web 2.0 (Schroth and Janner, 2007), which is a term to describe a recent tendency to design and implement interactive and collaborative Web sites enabled with rich support for social networking, paved the way for the emergence of the IoS (Buxmann et al., 2009). Supported with advances in the networking and mobile technologies making the Internet more and more ubiquitous (Bechmann and Lomborg, 2014), the IoS is a concept which describes the organisation of the future IT, where any resources needed for software applications are available online in the form of discoverable and easily accessible services. These resources include not just software assets, but additionally development tools and infrastructure to deploy and run this software, including servers, storage and networking support. Taken together, such a multi-level ser-

[1] MailCleaner, for example, offers a range of such services to enterprises, governments, Internet Service Providers (ISPs) and educational institutions (`http://www.mailcleaner.net/`).

[2] A classic example of an e-mailing service with built-in support for all kinds of malware detection is Google Mail (`https://mail.google.com/`).

vice provisioning model nowadays has become known as Cloud Computing.

2.1.1 Cloud computing as an extension of Service-Oriented Computing

SOC's benefits were attracting more and more enterprises seeking to implement their software systems following the SOA principles – that is, by minimising effort on implementing own software assets from scratch, but rather to re-use already existing resources via Internet. This attractive model has paved the way for Cloud Computing which has revolutionised the way enterprises and ordinary users can access computing, storage and networking resources nowadays. With its emergence, traditional computing has transformed into a service-based model, where resources are commoditised and delivered over a network, just like traditional utilities, such as water or gas (Buyya et al., 2013).

Even though we have been experiencing this revolutionary paradigm shift to cloud computing only in the last 10 year, the concept itself is not novel, and was first introduced back in the 60s by McCarthy (Abelson, 1999, Parkhill, 1966). However, it was not until recent advances in networking, hardware, mobile and virtualisation technologies in the last three decades (Armbrust et al., 2010) that eventually fulfilled this prophetic vision. In the 80s, IBM started releasing into mass production affordable personal computers available to ordinary users for home usage. This paradigm shift from huge industrial and academic mainframe machines to compact personal computers was also supported by Microsoft, who offered operating systems to make PCs even more ubiquitous and usable both at home and at work.

Then, in the 90s, the networking technology was eventually developed enough to provide sufficient bandwidth to support the emergence of the Internet and make it available to the masses (Mohamed, 2015). The Internet, together with new software interoperability standards, finally allowed enterprises to interconnect all their computers and opened many business opportunities to monetize this new computing paradigm. With the rise of commercial networking, enterprises started looking for novel mechanisms and service models for delivering their solutions and resources to end users via the Internet. In 1999, Salesforce.com[1] started delivering enterprise-class software through websites and became one of the pioneers in this field. Sales automation software offered by Salesforce.com could be accessed by customers via the Internet, and companies could purchase these services on a cost-effective and on-demand basis.

Next step was taken in 2002, when Amazon joined this trend and introduced

[1] http://www.salesforce.com/

its Amazon Web Services (AWS) – a cloud platform, which allowed users to access storage and computation resources, as well as some basic applications. In 2006, they went even further with the Elastic Compute Cloud (EC2), which offered an entire infrastructure to be delivered as a service to software developers – they could rent space to store and run their own applications in the cloud. It was 2009 when the cloud computing finally reached a stable level of maturity, and the cloud computing market was shaped with the main industry influencers getting on-board – IT giants like Google, Microsoft, IBM and Oracle were delivering their technological solutions to businesses and average users in the form of simple, accessible, on-demand cloud services.

As a result, cloud computing today "represents a concept of remote on-demand provisioning of pooled computing resources which are made available over a network, in a dynamic and scalable fashion, and whose consumption is metered to enable usage-based billing" (Kourtesis, 2011). Customers, ranging from individual users to entire organisations, need to pay cloud providers only when they access computing, storage or networking services. Additionally, they do not need to invest heavily in these potentially complex and expensive assets upfront, and then own and manage them in their premises, but instead always have them at their disposal and easily accessible via the Internet, and, consequently, reduce expenses on hiring a dedicated team of IT professionals.

Even though there exist several definitions of the term cloud computing, proposed in academic literature, with each placing emphasis on different aspects of the concept (Armbrust et al., 2010, Vaquero et al., 2008), the most intuitively comprehensive and widely adopted definition comes from National Institute of Standards and Technology (NIST) (Mell and Grance, 2009):

> *"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction."*

To provide an even more comprehensive explanation of what a cloud service is, this definition should be extended with a list of key characteristics a cloud service is expected to exhibit. Accordingly, NIST has distilled the following 5 essential features of a cloud computing service:

- **On-demand self-service** means that a user, subscribed to a cloud solution, should be provisioned with computing resources when needed automatically, without human interaction.

- **Broad network access** refers to the fact that all the cloud resources are accessible remotely over the Internet via standard interfaces and protocols.

- **Resource pooling** – thanks to recent advances in the area of virtualisation techniques, cloud resources are typically provisioned to customers in a multi-tenant and transparent manner. The former means that virtual computing resources belonging to different users (i.e., tenants) are hosted on a single physical server,[1] and the latter refers to the fact that these tenants are isolated from and unaware of each other. Moreover, they are also unaware of the exact geophysical location of their virtualised cloud resources, which can migrate from one physical host to another according to consumer demand.

- **Rapid elasticity of cloud resources** describes their capabilities to be (automatically) provisioned and released so as to enable scaling in and out and meeting ever-changing, dynamic requirements. Even though virtualised cloud resources of a particular cloud provider are constrained by the underlying physical infrastructure, from the consumer's perspective, this aspect of cloud computing creates an impression of seemingly infinite computing utility.

- **Measured service** is an inevitable consequence of SOC, where users pay only for the actual usage of a cloud service. It means, that a cloud offering is enabled with automatic monitoring, controlling and reporting tools to provide metering and billing in a transparent, fair and flexible fashion.

Even though cloud computing is typically associated with remote access to computing resources over the Internet,[2] for large enterprises consisting of multiple departments it is also possible to build up their own cloud data centre within their premises – this deployment model is known as 'private cloud'. The motivation behind this decision might be two-fold. Firstly, provided that an enterprise is large enough and its individual departments demand for elastic computational and storage resources which will be fully utilised, it can bring certain economic benefits when compared to expenditures needed for acquiring remote cloud services from commercial providers. Secondly, often migration to cloud is hindered

[1] Nowadays, it is also possible to reserve a whole cloud server so as not to share it with other tenants. For example, EC2 offers its subscribers so-called 'dedicated instances' (`http://aws.amazon.com/ec2/purchasing-options/dedicated-instances/`).

[2] This is essentially where the term 'cloud' comes from – in conceptual diagrams, a cloud-like shape was used to metaphorically denote the Internet in computer networks. This simplified representation also implied that the specifics of the internal organisation of the Internet are not relevant for the purposes of understanding the diagram. Similarly, nowadays end users are typically completely unaware of how and where a particular cloud service is actually deployed and operates.

with privacy and security issues – enterprises are not willing to release sensitive data beyond their private network perimeter. In such circumstances, it makes sense to deploy a private cloud locally, on premises. In this case, all potentially sensitive data is guaranteed to be stored and processed under the enterprise's control. A hybrid solution also exists – with this deployment model, some parts of the enterprise IT systems run on premises and the others – remotely in clouds. This way, enterprises can run critical computations under control and at the same time benefit from flexible pricing schemes by running less sensitive tasks in remote cloud environments.

When it comes to pricing schemes for a particular cloud service, in the first instance, it depends on the service model through which the resources are provisioned to users. Traditionally, NIST distinguishes between the following three service models (Mell and Grance, 2009):

- **IaaS** is a way of providing customers with on-demand access to processing, storage, networking and other fundamental computing resources (which are traditionally referred to as *hardware*) typically in the form of a virtual machine. IaaS users are then a required to install an OS and all necessary middleware and software so as to deploy, run and access arbitrary software over the Internet. Usually, users only have access to the OS, storage and deployed software, whereas management of the underlying hardware is beyond their capabilities and remains the responsibility of the IaaS provider. With the IaaS delivery model, cloud subscribers – usually companies who are attracted by the opportunity to save on acquiring and maintaining potentially expensive hardware – are charged for the consumption of the hardware resources. Typical monitored metrics for metering and billing are number of reserved computational instances and associated Central Processing Unit (CPU) and memory utilisation, storage consumption, network bandwidth, etc.

- **PaaS** model provides a computing platform equipped with a technological stack of solutions necessary to deploy and run users' software without the cost and complexity of buying, configuring and managing both the underlying hardware infrastructure and the computational platform. PaaS target groups are typically software vendors, IT departments of enterprises, individual developers, etc. – all those who want to shift a significant share of their concerns associated with developing, maintaining and provisioning on-demand software to the PaaS provider's end. This allows developers to focus on the business functionality of their software (which possibly can be provisioned as a SaaS offering), rather than spend their efforts on managing

the enabling cloud infrastructure and platform. The PaaS provider assumes the responsibility to monitor and manage the usage of every application and to allocate infrastructure resources as appropriate to meet ever-changing requirements (Kourtesis, 2011). Charging in PaaS also relies on metering the CPU and memory consumption, and additionally includes the consumption of platform-level services available to the users – for example, messaging queues or databases.

- **Software as a Service (SaaS)** is a delivery model in which customers' software deployed and running in the cloud environment is easily accessible from various devices (PCs, laptops, smartphones, etc.) and clients (i.e., rich and thin) in an on-demand manner. This delivery model is a move from the established practice of making software applications available as-a-product – that is, a distributed software product is installed and maintained by users (Kourtesis, 2011). Instead, the SaaS model relies on a providing a multi-tenant approach to software provisioning so as to facilitate having a single and shared cloud-based application code base, which can serve multiple customers. In these circumstances, customers' management capabilities are limited to configuring application-specific settings and managing personal business data; hardware infrastructure and platform resources, such as the OS or the database, are beyond their control, and remain a concern of the SaaS vendor (or a third-party PaaS provider in a case when cloud software is deployed on a third-party cloud platform).

2.1.2 Cloud Application Platforms delivering the Internet of Services

In recent years with the continued growth of the PaaS segment, more and more providers are offering software developers an extended set of functionalities and support not just for hosting and executing software assets, but also for developing, deploying, testing and maintaining – that is, with a support for a complete software life cycle. This typically includes Integrated Development Environment (IDE) plug-ins for easier deployment to the cloud environment, as well as monitoring, logging and reporting tools. Gartner refers to such deployment model as Application Platform-as-a-Service (aPaaS) and defines it as follows (Gartner, Inc., 2015):

> *"Cloud service that offers development and deployment environments for application services."*

This model, however, is not sufficient when an already existing application (coupled with other on-premises software components) has to migrate to the

cloud, or implement specific features beyond the standard capabilities of the programming framework (e.g., support for user authentication, e-mail notifications or queue messaging). In these circumstances, software developers are forced to implement these functional pieces of their application systems either by developing them from scratch or by deploying existing solutions on the cloud along with the main application. In either case, these components would only be accessed and used by a particular application, and application owners would be responsible for maintaining both the application and all the depending components. Though this approach provides users with full control and visibility into all pieces of the application system, it suffers from i) increased complexity associated with managing multiple components, ii) potentially low quality due to usage of custom-made software components instead of existing, reliable and optimised ones, iii) redundancy and excessive utilisation of storage resources due to the necessity to store individual, often duplicated components of an isolated application system on a cloud, and iv) relatively high time to market.

To address these shortcomings and to further facilitate the software development process, PaaS providers started compiling and packaging standard, frequently-used basic units of software functionality, repeatable across multiple applications, into ready-to-use software components. Then, these components were commoditised and offered through a 'pay-per-use' pricing model to cloud service subscribers, who could integrate them in their applications in a relatively effortless and declarative manner avoiding unnecessary source code modifications and tedious system configurations. For example, instead of deploying a separate queue messaging server on the cloud, users were offered to subscribe to an already running, reliable and highly-scalable solution offered by the cloud platform service marketplace. Similarly, instead of re-inventing the wheel and developing a potentially complicated and sophisticated authentication mechanism (e.g., with a support for Secure Shell (SSH) or database persistence), users could simply bind their application logic with an existing cloud component and concentrate on their immediate business-oriented tasks – that is, at a price of a small fee, save on the human and time resources.

Such reusable integrated components, or simply services, are a natural consequence of the SOC paradigm. Following the principles of SOC, PaaS providers offer their subscribers a wide selection of pre-existing and reusable services, ready to be seamlessly integrated into users' applications. These add-on services are typically provisioned to customers through a cloud service marketplace – a central catalogue of platform services and third-party extensions. By gathering multiple add-on services in one place and making them easily discoverable and accessible,

cloud platforms contribute to delivering a reliable IoS. By offering cloud services in such a way that software assets are assembled from existing components just like a Lego® construction set, such cloud platforms further reduce the human effort and capital expenses associated with developing complex software systems. This means that software developers – PaaS end users – can concentrate on their immediate, domain-specific tasks, rather than expend effort on, for example, developing their own authentication or e-billing mechanisms – instead, existing components are offered, managed and maintained by the cloud platform. The integration of users' applications with platform services usually takes place by means of APIs, through which software developers can easily couple necessary services with their applications and also perform further service management. To support rapid software development, such cloud solutions not only provision customers with an operating system and run-time environment, but additionally offer a complete supporting environment to develop and deploy SBAs, including a marketplace with a range of generic, reliable, composable and reusable services (Kourtesis et al., 2012, Rymer and Ried, 2011) (as illustrated in Figure 2.1). most prominent and commercially successful examples of such platforms include Google App Engine (38 services offered), Microsoft Azure (20 built-in services and 35 third-party add-ons offered), IBM Bluemix (61 services offered), and Heroku, the 'richest' cloud platform in terms of offered services, which comprises over 150 add-ons.

We call such cloud solutions as CAPs and, building on the Gartner's definition of aPaaS, provide the following definition:

> *"Cloud Application Platforms are a subset of PaaS offerings, which provide users with development and deployment environments, including a range of generic reusable software services."*

With this definition in mind, we have surveyed existing PaaS offerings to see how many of them qualify for the category of CAPs, and therefore can be seen as the targeted context of the research work presented in this thesis.

2.1.3 Survey of Cloud Application Platforms

The PaaS market is blooming – even though the main industry influencers like Google, Amazon, Microsoft, Oracle, IBM and Salesforce.com have already occupied the majority of the market (as of August 2015, their total market share is 67% (Finos, 2015)), new PaaS vendors still emerge trying to attract customers with novel appealing features. Gartner forecasts that the PaaS market will grow from $900M (in 2011) to $2.9B (in 2016) with aPaaS as the largest segment (Pettey and van der Meulen, 2015).

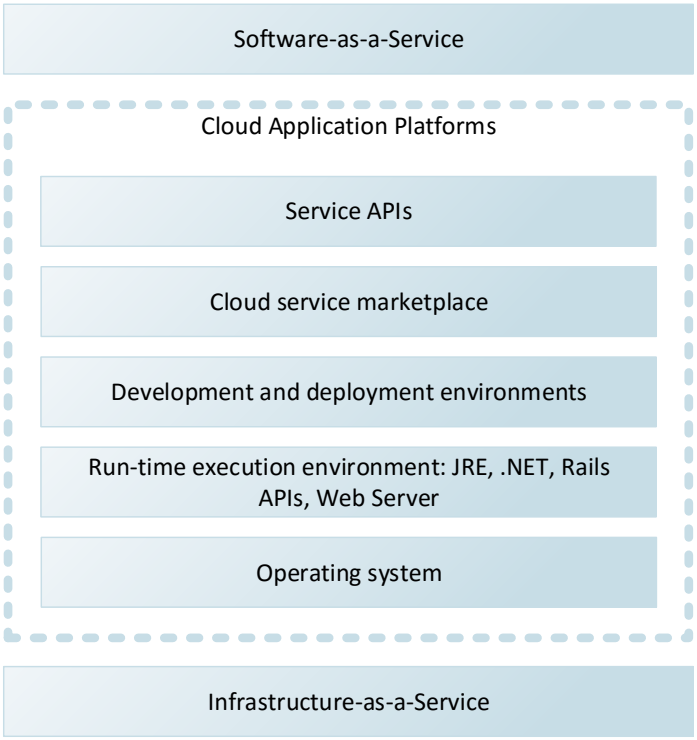

Figure 2.1: Besides an OS and run-time environment, CAPs offer a range of basic add-on services accessible via APIs.

We attempted to survey the existing market of commercially available public PaaS offerings aiming to identify the share of CAPs in the whole amount of existing solutions. It is worth mentioning that unambiguously classifying a cloud platform as a PaaS or IaaS offering is not so straightforward, as the difference might often be very subtle, or the actual description of a particular cloud offering is vague and misleading. For example, PagodaBox[1] exhibits all the characteristics of PaaS, even though it describes itself as just a 'hosting framework'. Same applies to the distinction between SaaS and PaaS – for example, Zoho[2] is typically positioned as a SaaS cloud provider, offering its users customisable Customer Relationship Management (CRM) and office software solutions as a cloud service, known as Zoho Office Suite. However, it also delivers the Zoho Creator platform, which can be classified as PaaS, because it allows creating new applications through drag-and-drop mechanism, and therefore was included in our survey. Another point of consideration for us was whether to include 'vertical' PaaS solutions – that is, cloud platforms which offer services to a very narrow and

[1] https://pagodabox.com/
[2] https://www.zoho.com/

specific domain. For example, Appistry[1] is a PaaS solution mainly focusing on the biomedicine domain. We have decided not to include such cloud platforms due to their limited usage. In our survey we also omitted open-source frameworks for deploying cloud platforms on private or rented premises – these include, for example, Apache Stratos,[2] AppScale,[3] or OpenShift Origin.[4]

The main three aspects taken into consideration when classifying PaaS products as CAPs are:

1. **Tooling support for software development process throughout the whole lifecycle** – that is, developing, testing, deploying, and maintenance. This can include such features as IDE plugins for easier and faster deployment, remote access to cloud servers via the File Transfer Protocol (FTP) and SSH, Secure Sockets Layer (SSL) encryption for increased security, support for software versioning (e.g., Subversion (SVN), Git, Mercurial), online management and monitoring consoles for observing applications' health and performance, etc. Another specific feature offered by some of the cloud platforms (e.g., Caspio, webMethods AgileApps Cloud, Force.com and Zoho Creator) is the possibility to create applications through the point-and-click interface, which exempts software developers from manual coding and thus saves considerable time resources. By simply dragging and dropping required components users can create applications in minutes.

2. **Support for multiple programming languages** is another typical feature of cloud platforms. Most of them support the most commonly adopted languages for Web development, including, PHP, Java, Ruby, Python, Node.js, .NET, Perl, Go, as well as an extensive range of software development frameworks, such as Java Spring,[5] CakePHP,[6] PHP Zend,[7] or Ruby Sinatra.[8] Some of the platforms, offering the point-and-click development model, are using their own proprietary languages like Apex (Force.com) and Deluge Script (Zoho Creator) to execute visually designed applications. Other platforms, such as CatN, Cloudways, Fortrabbit, PagodaBox and Evia Cloud, speak only one language – PHP, and essentially extend the traditional PHP hosting with scalable cloud infrastructure.

[1] http://www.appistry.com/
[2] http://stratos.apache.org/
[3] http://www.appscale.com/
[4] https://www.openshift.com/products/origin
[5] https://spring.io/
[6] http://cakephp.org/
[7] http://framework.zend.com/
[8] http://www.sinatrarb.com/

3. **Presence of a service marketplace**, through which basic software services, provided by platform providers or third parties, can be integrated into users' applications, is the primary criteria for us when identifying CAPs. CAPs may use different terms to refer to basic services offered (e.g., add-ons, plug-ins, features, extensions, apps, etc.), but the concept is still the same – CAPs offer its customers a way of integrating existing units of software functionality into applications through standard APIs and pay-per-use subscriptions.

As a result, in our survey we identified 24 different PaaS offerings, out of which 18 can be classified as CAPs, which makes up 75% (the detailed survey can be found in Appendix C). These numbers give us confidence that the problems, raised and tackled by the presented research work, are relevant, and the solution we are developing has the potential to be applied across a large number of cloud platforms.

2.2 Autonomic Computing

Today's IT has been steadily growing in size and complexity, and nowadays consists of complex computing systems and operates in highly-heterogeneous distributed environments. With proliferation of mobile devices and advances in cloud computing, networking, and virtualisation, modern computing systems have reached the maturity level, which lies beyond humans' manual capacities to maintain them at a stable and operational level. Indeed, effectively managing the underlying infrastructure of modern computing systems requires from enterprises to exercise constant and timely control over hundreds of components and thousands of tuning parameters (Muller, 2006) – this task, if addressed only by human operators in a manual manner, may be associated with unaffordable human or time resources. While the cost of technology was constantly dropping, investing in skilled management and administration personnel was constantly rising (IBM Corporation, 2005). Therefore, a logical and obvious decision in these circumstances was to offload these tasks from humans by automating routine management processes as much as possible.

To some extent, a similar solution was applied in the 1920s in telephony (Huebscher and McCann, 2008) when automatic branch exchanges were introduced to cope with the increased telephone usage and to replace human operators. Today, a similar goal is pursued by the Autonomic Computing research, which sets its goal to minimise the role of human administrators in operation and management of complex computing systems. First suggested by IBM (Horn, 2001) in 2001, the term of Autonomic Computing introduced a model for computer sys-

tems that are enabled to manage themselves with minimum human involvement. Today, Autonomic Computing is a concept that brings together many fields of IT (i.e., Control Theory, Agent Theory, Software Engineering, Information and Data Management, etc.) with the purpose of creating computing systems capable of self-management. Accordingly, IBM compared complex computing systems to the human body, which is able to regulate unconscious bodily functions such as respiration, digestive functions, pupil adjustments, etc. by means of the autonomous nervous system. From this perspective, computing systems should also exhibit certain autonomic properties and be able to independently take care of the regular maintenance and optimisation tasks, thus reducing the workload on the system administrators (Huebscher and McCann, 2008). By constantly sensing the surrounding context and responding to changes, autonomic technology can enable systems to perform a wide range of self-governance tasks, thus lessening the burden on the IT staff to initiate and handle those tasks (IBM Corporation, 2005).

A possible way of enabling computing systems with self-managing capabilities is through *self-reflection*. A self-reflective system uses a causally connected self-representation – a model of its internal structure and its surrounding environment – to support self-monitoring and self-adaptation activities (Blair et al., 2004). In other words, such a system is self- and context-aware, which enables it to perform run-time adaptations, so that applied changes dynamically reflect on the state of the system (thus, possibly, triggering another adaptation cycle) (Dautov et al., 2012). The rationale behind self-reflection is to equip computing systems, deployed in hostile and/or dynamically changing environment, with sufficient knowledge to enable them with capabilities to react to various changes in the surrounding context, based on that knowledge. Such scenarios are typically characterised with a predominant role of time constraints – in these circumstances the capability of a remote system to perform automatic adaptations at run-time on its own, without the necessity to wait for a human operator, within a specific time frame, is often of a great importance. The concept of self-reflection can be summarised in the following characteristics[1] (Nami and Sharifi, 2007):

- **Self-awareness** of an autonomic system includes knowledge of its internal structure, relationships between sub-components, available resources, etc.
- **Context-awareness** of an autonomic system is an ability to be aware of what

[1]Often in the literature researchers use their own terms to refer to the same concepts introduced by Horn (e.g., self-tuning and self-assembly are similar to self-configuration). There are also research works which present their own views on the autonomic systems, different from, yet inspired by IBM's vision. In (Cavalcante et al., 2015), for example, self-awareness is used as an umbrella term covering three properties: self-reflection, self-prediction, and self-adaptation. Throughout the rest of this document we will be using the established terms as presented in this section.

is happening in the surrounding execution environment through observation and interpreting these observations.

- **Openness** refers to an ability of an autonomic system to operate in heterogeneous environments and be portable across multiple platforms.

- **Anticipatory** behaviour of an autonomic system means that it should be able to estimate the amount of required resources in order to meet emerging user requirements, in a transparent manner, keeping the complexity hidden.

Demonstrating these four characteristics for a self-reflective system would not be possible without some sort of a self-representation knowledge repository, which would provide all the necessary information to enable computing systems with self-reflective features. To facilitate self-awareness and context-awareness, this knowledge base has to be populated with information about the internal structure and organisation of the autonomic component, as well as its environment and context and possible ways of perceiving them. To support openness, it has to include information about how to connect and interact with other elements in various environments. Finally, to demonstrate anticipatory behaviour, it is important to have an extensive set of policies and rules determining the ability of the system to predict various changes. As explained in the next section, this self-reflective knowledge component plays a key role in implementing closed feedback loops when engineering autonomic systems.

2.2.1 MAPE-K reference model

IBM's vision of Autonomic Computing is structured around a reference model for autonomic control loops (Huebscher and McCann, 2008), known as the MAPE-K (Monitor – Analyse – Plan – Execute – Knowledge) adaptation loop, and depicted in Figure 2.2. Stemming from Agent Theory, the MAPE-K model can be seen as an evolution of the generic model for intelligent agents proposed in (Russell et al., 2010). According to this model, agents are enabled with sensors to perceive their environment, and are able to execute certain corrective actions on the environment based on the observed values. This continuous process of sensing and acting upon sensed values clearly corresponds to the closed adaptation loop of the MAPE-K model. Applying the model to the domain of self-management at the PaaS level, we now consider each of its elements in more details.

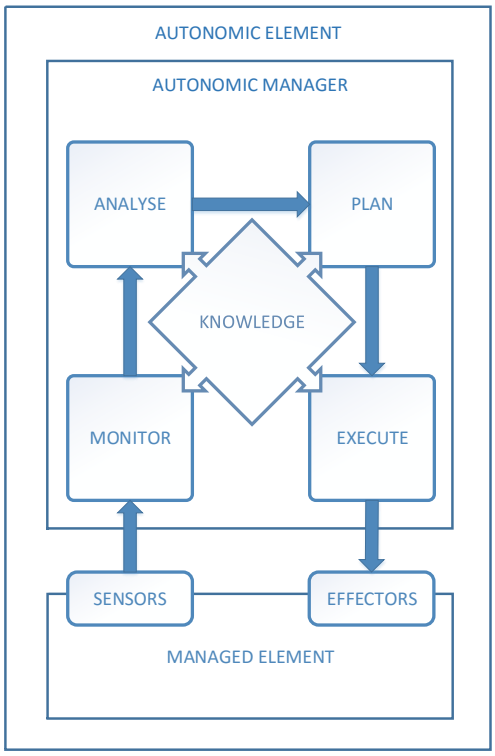

Figure 2.2: IBM's MAPE-K reference model for autonomic control loops (Kephart and Chess, 2003).

Managed element

Managed elements represent any software and hardware resources which are enhanced with autonomic behaviour by coupling with an autonomic manager. In the context of clouds, the managed element may be a cloud platform as a whole, a web server, a virtual machine, an operating system, etc. In the presented research work, as explained in Chapters 5 and 6, primary managed elements include applications deployed on a CAP, and add-on services offered to users through the CAP marketplace. Managed elements are equipped with sensors and effectors. The former, also known as probes or gauges, are software or hardware components responsible for collecting information about the managed elements. Sensors are typically associated with metrics – certain characteristics of the managed element, which need to be monitored (e.g., response time, memory utilisation, network bandwidth, etc.). The latter are hardware or software components, whose responsibility is to propagate required adaptation actions to the managed element. Depending on the scale, adaptations can be coarse-grained (e.g., completely substituting a malfunctioning Web service) or fine-grained (e.g., re-configuring that

service to fix it).

Autonomic manager and knowledge

The autonomic manager is the core element of the MAPE-K model – essentially, this where the whole MAPE-K functionality is encapsulated. This software component, configured by human administrators to follow high-level goals, uses monitored data received from sensors and internal (i.e., self-reflective) knowledge of the system to analyse these observations, plan possible adaptation actions if needed, and execute them to the managed element through effectors so as to achieve those goals.

The internal self-reflective knowledge base of the system, shared between the four MAPE-K components, may include any type of information required to perform the Monitor-Analyse-Plan-Execute (MAPE) activities. In the first instance, in includes an architectural model of the managed element – a formal representation of its internal organisation, subcomponents, and connections between them. Another important component is a set of diagnosis and adaptation rules – formally defined policies, which serve to analyse critical situations and choosing a relevant adaptation plan among existing alternatives. Among other things, the knowledge base can also include historical observations, data logs, repositories of previously detected critical situations and applied adaptation solutions, etc., which implies that the knowledge base is not designed to be static (i.e., populated once by administrators at design-time), but rather has to evolve dynamically over time by accumulating new information at run-time. In connection with this, there is room for applying certain techniques from Machine Learning research, and we describe this possibility in more details in Section 9.2.

Monitoring

The monitoring component of the MAPE-K loop is responsible for gathering information about the environment, which is relevant to the self-managing behaviour of the system. In a broad sense, monitoring may be defined as the process of collecting and reporting relevant information about the execution and evolution of a computer system, and can be performed by various mechanisms[1] (Kazhamiakin, Benbernou, Baresi, Plebani, Uhlig and Barais, 2010). These monitoring processes typically target the collection of data concerning a specific artefact, the monitored subject (Bratanis et al., 2012). In the context of CAPs, monitored subjects

[1] In our research we focus on run-time monitoring. Related activities can also include such techniques as post-mortem log analysis, data mining, and online or offline testing – the interested reader is referred to (Kazhamiakin, Benbernou, Baresi, Plebani, Uhlig and Barais, 2010).

include platform components, deployed applications, service compositions, individual services, etc., and monitoring properties can be the number of simultaneous client connections, data storage utilisation, number of tasks in a messaging queue, CPU and memory utilisation, response times to user requests, I/O operations frequency, etc. Appropriate monitored data helps the autonomic manager to recognise failures or sub-optimal performance of the managed element, and execute appropriate changes to the system. The types of monitored properties, and the sensors used, will often be application-specific, just as effectors used to execute changes to the managed element are also application-specific (Huebscher and McCann, 2008).

Two types of monitoring are usually identified in the literature:

- **Passive monitoring**, also known as non-intrusive, assumes that no changes are made to the managed element. This kind of monitoring is typically targeted at the context of the managed element, rather than the element itself. For example, in order to monitor some metrics of a software component, in Linux there are special commands (e.g., `top` or `vmstat` return CPU utilisation per process (Huebscher and McCann, 2008)). Linux also provides the `/proc` directory, which contains runtime system information, such as current CPU and memory utilisation levels for the whole system and for each process individually, information about mounted devices, hardware configuration, etc. (Huebscher and McCann, 2008). Similar passive monitoring tools exist for most operating systems. As we will further explain in more details, in our use case, to collect information about database space occupied by a table, we did not measure the size of that table *per se*, but rather queried a special separate table, whose role in relational databases is to keep live and up-to-date statistical data about the database. This is another example of passive monitoring. The drawback of this approach, however, is that often monitored information is not enough to unambiguously reason about possible sources of problems in the system.

- **Active monitoring** assumes designing and implementing software in such a way that it provides an API with some entry-points for capturing required metrics and collecting monitored data. For example, to monitor response time from an external web service, an application should provide an API in order for the monitoring component to intercept the requests. This can often be automated to some extent. For instance, using aspect-oriented programming techniques allows 'injecting' additional functionality into the programme source code. Moreover, there are some tools for inserting probes

into already compiled Java byte code (e.g., ProbeMeister[1]), which makes it possible to actively monitor legacy systems. This type of monitoring is also known as intrusive, as it inevitably implies making changes (i.e., intrusions) to the managed element by instrumenting it with probes to facilitate inspection of its characteristics. As with code instrumentation, it is essential that this is done with care, since the instrumentation can itself potentially affect the subject's performance, providing a flawed picture of its inherent capabilities.

It is also important to consider who is responsible the monitoring process and data collection. Typically there are two data collection methods (or a combination of them) used (Bratanis et al., 2012). In polling mode, the autonomic manager is responsible for querying the managed element and its sensors at regular intervals. On contrary, in push mode it is the managed element's responsibility to notify the monitoring authority of any significant events or changes. In practice, implementing monitoring as part of the self-management functionality may require a deliberate and flexible combination of several approaches. In the context of CAPs, on the one hand, it can be a condition of deployment that add-on services and user applications must conform to a platform-specific API and expose some sort of 'hooks', to which monitoring sensors will be attached, thus implementing an intrusive approach to data collection. This in turn can be seen as a potential security threat, since such kind of hooks imply providing access to potentially sensitive internal application data, which may be unacceptable as far as particularly sensitive software systems and business data are concerned, and therefore a less or non-intrusive approach would be preferred.

Moreover, a malfunctioning application or a service cannot be relied upon to act according to their designed specification – that is, it is potentially risky to assume that a report signalling its own failure will be pushed to the monitor by the managed element. In these circumstances, it is important that a PaaS-level autonomic manager implements 'heart beat monitoring' and polls managed elements at regular intervals to check whether they are still alive and active. On the other hand, however, the SOC paradigm suggests software systems are often combined in novel, emerging and unpredicted ways, which makes it impossible to make the autonomic manager aware of all of the possible situations in advance. In such circumstances, applications are also required to be able to push event notifications to the autonomic manger, since it may not itself issue the necessary requests.

As self-managing systems grow and the number of sensors increases, moni-

[1] http://www.objs.com/ProbeMeister/

toring activities may result in a considerable performance overhead. That is, in a system with thousands of probes constantly generating values, the monitoring component may not be able to cope with this overwhelming amount of data. To avoid 'bottlenecks', system architects have to distinguish between values which are relevant to self-managing activities and so called 'noise' – data, which can be neglected. Another potential solution to this problem is performing high-level monitoring first, and then, once an anomaly is localised, activate additional monitoring resources (Ehlers et al., 2011). With this approach, computational resources are provisioned to the monitoring component on-demand, only when a problem is detected, thus resulting in a higher efficiency and resource consumption.

Analysis

The analysis component's main responsibility is to perform current situation assessment and detect failures in the managed element. In its simplest form, the analysis engine, based on Event-Condition-Action (ECA) rules, simply detects situations when a single monitored value is exceeding its threshold (e.g., CPU utilisation reaches 100%), and immediately sends this diagnosis to the planning component. However, the problem determination may be a challenging task, especially in a distributed environment when the monitored data is coming from multiple remote sources. Based on the internal knowledge, the autonomic manager should decide whether a particular combination of monitored values represents or may lead to a failure.

In connection with this, techniques from the area of Complex Event Processing (CEP) research (Margara and Cugola, 2011) proved to be helpful in the context of data analysis and situation assessment. From CEP's point of view, everything happening in the environment and changing the current state of affairs is an atomic event. Sequences of atomic events build up complex events, which, in their turn, may be part of an even more complex event, thus building event hierarchies. For example, when CPU and memory utilisation levels of several VMs running on the same physical machine reach 100% (i.e., atomic events) within a short period of time, this indicates that the utilisation of the whole physical machine has reached its limit (i.e., the complex event). In Chapters 5, 6 and 7 we will explain in more details how CEP techniques were utilised in our own research work.

2.2.2 Levels of autonomicity

Modern IT still heavily depends on human operators, and the industry is still away from pervasive implementation of the autonomic computing principles. However,

ever-growing computing systems are getting more and more complex, and there is a clear need for enabling them with autonomic behaviour and capabilities for self-adaptation. The interest in autonomic computing is growing, and the industry is taking an evolutionary approach to improving existing computing systems by enabling them with self-managing capabilities (Ganek and Corbi, 2003). It means that certain bits of computing systems are gradually replaced or enhanced by autonomic components so as to eventually achieve fully autonomic behaviour. In our work, we refer to the hierarchy of autonomic behaviour, proposed in (Ganek and Corbi, 2003) and summarised in Table 2.1, which suggests 5 steps towards fully autonomic behaviour.

Table 2.1: Levels of autonomic behaviour (modified from (Ganek and Corbi, 2003).

	Basic level	**Managed level**	**Predictive level**	**Adaptive level**	**Autonomic level**
Characteristics	Multiple independent components generate raw data to be collected, correlated and analysed.	Heterogeneous data is collected and synthesised in a single place by means of dedicated management tools.	The whole system is able to monitor and analyse data, and suggest possible adaptation actions to operators.	System is able to perform data analysis, adaptation plan generation and execution of the generated plan on its own.	Driven by the set of self-management policies, the system is able to diagnose critical situations, choose an adaptation plan from several existing alternatives, and successfully apply it.
Role of human operators	Extensive, highly-skilled IT staff is responsible for initial collecting and interpreting data, and further planning an execution of adaptation plans	IT staff needs to analyse data to generate and execute appropriate adaptation plan	IT staff approves and initiates adaptation actions suggested by the system	The role of the IT staff is supervisory; minimum number of human operators is only required to monitor system performance with respect to a set of SLAs	IT staff focuses on enabling business needs, and is only required to change self-management policies if needed.

MAPE implementation	N/a	Monitoring	Monitoring, Analysis, and Planning	The whole MAPE cycle is implemented (the system operates with minimum human supervision)	The whole MAPE cycle is implemented (the systems operates without human involvement)
Benefits	N/a	Increased system awareness; improved productivity and decreased time, required to apply adaptation actions to the system	Reduced dependency on highly-skilled professionals; decreased time required to apply adaptation actions to the system	Minimum dependency on extensive IT staff involvement; increased business agility, flexibility and resiliency	No dependency on extensive IT staff; business policies drive the IT management; increased business agility and resiliency.

- The **basic level** represents the starting point for most of the IT systems which were not designed to be enabled with self-managing capabilities in the first place. At this level, various heterogeneous system components independently from each other generate raw data, which is then manually collected, analysed and transformed into adaptation actions by IT professionals. Accordingly, all the management tasks, such as system configuration, optimisation, healing, protection, are performed manually. These challenges require the IT staff to be highly experienced and skilled.

- At the **managed level**, which is a first step towards full autonomicity, an integrated approach to system monitoring is applied, and monitored data coming from independent heterogeneous system components is gathered in one place, providing human operators with a more holistic view on the current state of the whole system. This reduces the time for operators to collect and correlate data across the whole system.

- At the **predictive level**, individual system components are able to monitor themselves, analyse and assess situation and context, recognize patterns, and suggest optimal configurations for human operators, who need to approve and apply these changes.

- At the **adaptive level**, the system, equipped with necessary self-reflective knowledge and set of diagnosis and adaptation policies, is able to detect and

diagnose potentially critical situations, and automatically take necessary actions to perform self-adaptations. The knowledge base, among other things, also includes a set of Service Level Agreements (SLAs), which primarily drive the operation of the system and its self-management. The involvement of human operators at this level is minimal, and is only required for supervision purposes.

- Finally, at the fully **autonomic level**, the system operation is solely governed by business policies and objectives, established by human administrators at design time. The IT staff can concentrate on immediate business-related tasks, and is only required whenever there is a need to modify the self-management policies (though altering business policies can also be performed by non-IT professionals).

When applying the presented self-management hierarchy to the research work presented in this thesis, we can classify it under the upper three levels – predictive, adaptive and autonomic. Even though the EXCLAIM framework developed within the context of this research work mainly focuses on the monitoring and analysis activities of the MAPE-K cycle, it is expected to facilitate further planning and adaptation processes to be performed i) manually – at the predictive level, ii) automatically under minimum human operator supervision – at the adaptive level, or iii) automatically without human involvement – at the fully autonomic level. In Chapter 7 we describe our experience of enabling CAP application owners with a capability to detect and diagnose situations when the resource utilisation is approaching its critical threshold. Furthermore, in Section 9.2 we also discuss the potential of integrating the EXCLAIM framework with existing planning and execution mechanisms, thus aiming at creating an autonomic framework fully implementing the MAPE-K functionality.

2.2.3 Self-management in clouds

To some extent, SOC addressed the problem of complexity in ever-growing and expanding computing systems. As the number of individual components constituting enterprise systems was growing exponentially, their timely management was becoming an increasingly challenging task. Figure 2.3 schematically illustrates this basic concept. With the traditional software architecture, enterprises, existing in isolation from each other, need to develop their own components from scratch and manage them afterwards. With the SOA approach, there is no need to maintain redundant components – each of the enterprises is in charge of a single software component, whereas the other components constituting their application

systems are provisioned by other enterprises as services. As this simple diagram illustrates, at a small price of additionally managing network communication, the SOC model allowed reducing software management efforts by a factor of N, where N=4 is the number of enterprises and services constituting their SBAs.

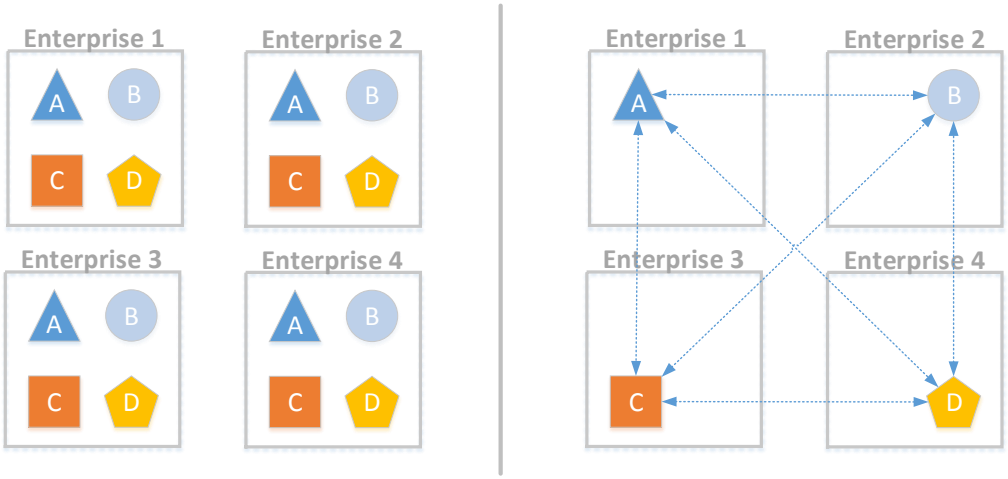

Figure 2.3: Traditional multi-component enterprise software architecture and service-oriented architecture.

With the continuing paradigm shift towards cloud computing, service-based cloud environments are expected to become even more complex. The diagram in Figure 2.4 schematically depicts how cloud computing follows the SOC model and accumulates multiple IT services available to users in one place. By providing a selection of IT resources offered as services to customers, on the one hand, they exempt them from tedious and often unnecessary tasks of IT management. On the other hand, however, the burden of IT management does not simply vanish – it is now the cloud provider's responsibility to maintain the required Quality of Service (QoS) and meet SLAs.

Given the fact that the number of available cloud services is constantly growing, and their timely management is outgrowing cloud providers' capacity to manage them in a responsive manual manner, we are running into a risk of ending up in a situation where service-based cloud platforms become tangled, complex, and unreliable environments. Both academia and industry are coming to realise this problem and consequently putting considerable effort into finding potential solutions to address it. Attempts to enable clouds with self-managing mechanisms have mainly focussed on the infrastructure level. Mechanisms for distributing the varying volumes and types of user requests across different computational instances (load-balancing), or by reserving and releasing computational resources

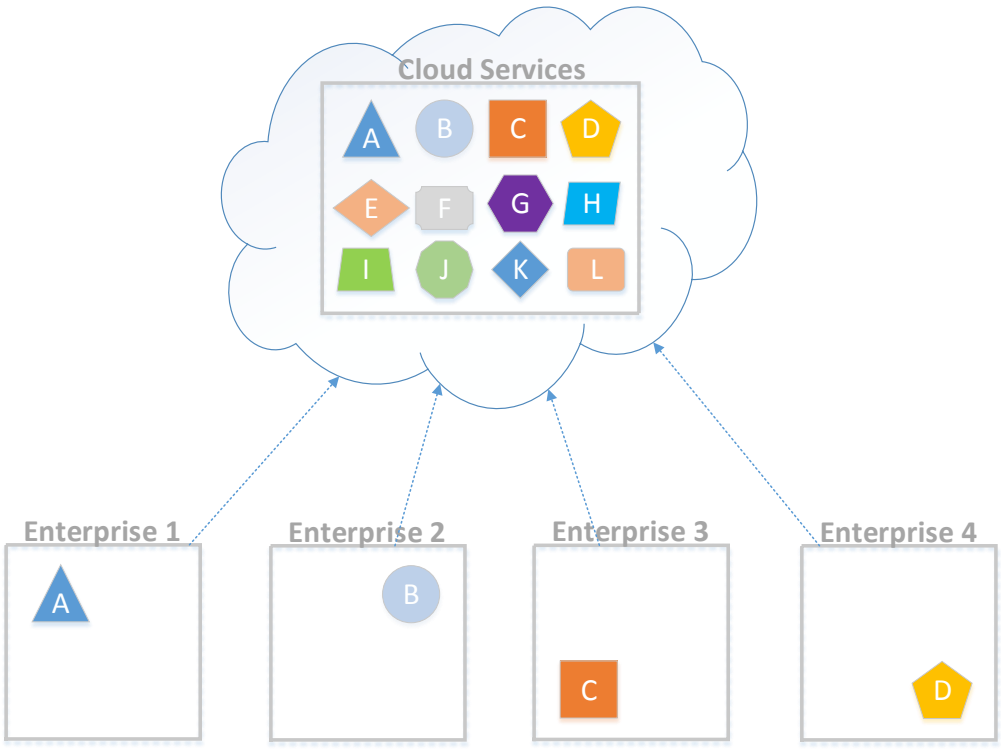

Figure 2.4: Clouds follow the SOC model by gathering IT services in one place.

on demand (elasticity) (Armbrust et al., 2010, Natis et al., 2009) are nowadays seen as 'must-have' features of a cloud offering. In both cases, self-adaptation mechanisms typically implement a closed control loop for self-adaptations, where main monitoring metrics are CPU and memory utilisation of computational instances (i.e., virtual machines), and network bandwidth. More specifically, to facilitate load balancing, IaaS providers observe current resource consumption and take necessary steps to move computational instances – that is, to spread users' workload – across multiple physical servers. By doing so, they aim at achieving even workload distribution across their datacentres and avoiding situations when some servers are overloaded with data storage and intensive computations, while others stay idle. As for elasticity, IaaS providers monitor customers' resource utilisation and are able to automatically take necessary steps to either scale down by shutting down excessive instances associated with a particular customer, or scale up by launching additional instances in response to reaching a critical level of resource utilisation. As a result, customers benefit from both types of activities – they do not need to pay for idle instances (that is, for the resources they are not actually using), and are also secured from their applications crashing due to the

shortage of resources. Both load balancing and elasticity are now seen as essential characteristics of cloud computing (Mell and Grance, 2009).

In some cases, the process of adding/removing computational instances is managed by the cloud platform in a completely automated way (e.g., in OpenShift, Google App Engine, or AWS Elastic Beanstalk). In other cases, application developers are first expected to configure such capabilities by specifying critical thresholds when resources have to be provisioned/de-provisioned. Other cloud platforms only provide an API for performing such activities (e.g., CloudFoundry or Heroku), and users are asked to monitor their resource utilisation and accordingly apply changes programmatically. Third-party solutions have also proved to be useful when automating the task of scaling applications up/down (e.g., a commercial offering HireFire[1] for Heroku). Another relevant technique offered by several platforms, including OpenShift and Heroku, is to idle inactive computational instances based on users' recent activity to save cloud platform resources and users' money.

A basic technique employed by most of the existing cloud self-management mechanisms is 'heart beat monitoring' which relies on sending polling request to the monitored components within certain time intervals to check if it is still 'alive' (that is, responding to these requests), and, if not, taking necessary steps to restart it (for example, see Dead Man's Snitch for Heroku.[2]) Self-management behaviour of this kind is rather simple, and does not involve any sophisticated analysis of what the underlying problem might be and how to solve it. Applications are treated as 'black boxes' and usually, if they still do not operate after several attempts to restart, they are fully stopped and further actions need to be taken by users. To support more in-depth monitoring of deployed applications and provide customers with visibility into resource utilisation and operation performance, some cloud platform operators either offer built-in monitoring tools (e.g., Amazon CloudWatch[3] or Google App Engine Dashboard) or employ external monitoring frameworks[4] (e.g., New Relic[5] and Sensu[6]). Common monitored metrics are CPU and memory utilisation, disk reads and writes, network traffic, number of incoming requests, etc. However, such monitoring frameworks often only deal with collecting and displaying data, and do not provide any means of automatically recognising and fixing a problem once it appears – this task is left

[1] https://www.hirefire.io/
[2] https://elements.heroku.com/addons/deadmanssnitch/
[3] http://aws.amazon.com/cloudwatch/
[4] Please refer to Chapter 4 and Appendix C for a more detailed overview.
[5] http://newrelic.com/
[6] http://sensuapp.org/

to the administrators. In this sense, existing solutions for supporting autonomic behaviour can only qualify for the managed level of self-management (please refer to Table 2.1). In Chapter 4, we provide a more detailed and critical overview of the existing solutions in the domain of autonomic cloud platforms and associated data monitoring and analysis solutions.

2.3 Summary

In this chapter, we familiarised the reader with the cloud computing paradigm, and its main underpinning concept – SOC. This explanation puts forward the motivation behind the presented research, and outlines its main context – namely, CAPs and their add-on service marketplaces. To support relevance of our research, the chapter also surveyed existing PaaS offerings and identified 75% of CAPs among them. The challenge of insufficient capabilities for self-governance is expected to be addressed by autonomic computing, which is also presented in this chapter. The chapter provided a classification of self-managing computing systems with respect to the extent, to which they implement the autonomic computing principles and minimise human involvement. It also explained the MAPE-K reference model for implementing self-adaptations in computing systems. Understanding this model is important to follow the description of the EXCLAIM framework, which was designed and implemented in the context of this research work and is explained in Chapters 5, 6, and 7.

Chapter 3

Related technologies: Big Data processing and the Semantic Sensor Web

In the previous chapter we explained the notion of Autonomic Computing and highlighted the role of an extensive knowledge base in implementing closed autonomic loops in complex computing systems. Often, in the context of timely data monitoring and analysis, to transform data into information, it is necessary to process considerably large amounts of raw heterogeneous data, which may go beyond traditional processing and storage capabilities. Accordingly, this chapter introduces the notions of Big Data and existing processing and storage techniques for coping with these extreme volumes of data. The chapter first defines Big Data focussing on its main four aspects, known as the 'four Vs' – namely, Volume, Velocity, Variety and Veracity. Then it continues with an overview of existing Big Data storage and processing techniques. Given the fact that in the context of CAP data monitoring and analysis, timely reaction to rapidly changing observations is of paramount importance, a separate section is dedicated to explain how the Velocity aspect of Big Data can be addressed. It explains the notions of 'data in motion' and stream processing, and introduces IBM InfoSphere Streams – a platform specifically designed for handling Big Data in motion, which was utilised in the presented research work, as explained in Chapter 7.

However, stream processing on its own cannot cope with data variety and heterogeneity. In scenarios, where data is represented using multiple formats and syntaxes, querying streams becomes problematic. Until recently, the Sensor Web had been one of such problem domains. To address this issue, the SSW community introduced a shared semantic vocabulary of terms to represent heterogeneous

data in a uniform way and developed appropriate technological solutions for handling such semantically-annotated streaming data. There are certain similarities between sensor networks and the domain of CAPs. By drawing parallels, we expect to benefit from existing techniques in this research field with a goal to overcome data heterogeneity (i.e., the Variety aspect of Big Data). Accordingly, the second part of the chapter first introduces the domain of traditional physical sensor networks, and then proceeds with an explanation of what benefits semantic technologies can provide. We explain the concept of the SSW and its main ingredients – namely, the Semantic Web technology stack, the SSN ontology, and RDF stream processing – in more details.

3.1 Big Data processing

Recent advances in various fields of IT, including sensor and network technology, social networking and cloud computing, have resulted in a situation, where the world creates an overwhelming amount of raw data on a daily basis. These unprecedented avalanches of data come from many sources – wired and wireless sensory readings, mobile data, stock exchange trade information, dynamic content from social networks such as Twitter, Facebook and YouTube, scientific data, and many others. According to IBM (IBM Corporation, 2015), the overall amount of data generated by the world every day equals to 2.5 exabytes (EB),[1] whereas in 2011 only, the total volume of data, created or copied, reached 1.8 zettabytes (ZB),[2] and since then this number has increased by nearly 9 times (Gantz and Reinsel, 2011). It is also predicted that by 2020 the world will generate 50 times the amount of currently existing data. At the same time, IT staff needed to manage this data will only grow less than 1.5 times (Gantz and Reinsel, 2011). In these circumstances, the scientific community urgently had to address multiple research questions, which concern various aspects of handling these extreme amounts of data, which have become known as Big Data. In the first instance, these research questions arise from how Big Data has to be analysed, captured, searched, shared, stored, transferred, secured, and visualised.

The term 'Big Data' itself was first coined in 1997 by Cox and Ellsworth (Cox and Ellsworth, 1997) to refer to datasets of around 100 gygabyte (GB) in size. What is now considered to be a norm and can easily be stored on a single Universal Serial Bus (USB) stick, at that time was large enough to make the storage and computing requirements go beyond capabilities of traditional approaches existing

[1] 1 EB = 10^{18} bytes
[2] 1 ZB = 10^{21} bytes

at that time. Since then, the amounts of generated data have grown immensely, and today the volume of a big dataset typically ranges from several terabytes (TB)[1] to several EBs (Manyika et al., 2011) depending on the context – due to these fuzzy and uncertain borders, there is still no common and precise understanding of what Big Data is. Nevertheless, in this document we will be using the term Big Data in a rather broad sense, and will define Big Data as "the datasets that could not be perceived, acquired, managed, and processed by traditional IT and software/hardware tools within a tolerable time" (Chen et al., 2014). The notion of 'traditional IT and software/hardware tools' typically includes conventional Relational Database Management Systems (RDBMSs), whose forte is in storing and querying data in the relational format – that is, as collections of related data held in tables, consisting of rows and columns. However, the relational format and structured nature are not necessary the case with Big Data, which is characterised with its high heterogeneity and unstructured or semi-structured nature. Strong points of RDBMSs become their limitation, which either make them completely inapplicable for Big Data scenarios or cancelled potential benefits, such as fast querying or indexing, and resulted in situations where resources invested in scaling up existing RDBMSs were disproportionate to the outcome. As RDBMSs were becoming more and more demanding in terms of expensive hardware, in order to cope with the increasing pressure of Big Data processing, the community started exploring other options to address this challenge. In the next section of this chapter, we will present a brief overview of the existing solutions.

Even though the Big Data hype started with its extreme size, which was the primary reason for introducing novel technological solutions, nowadays the concept of Big Data is multi-faceted and typically includes the following key features, widely known as 'the fours Vs' of Big Data.

Volume: extreme scale of data created and stored

The volume aspect was (and still is) the primary concern of the Big Data consortium. As of 2012, the world creates about 2.5 EB of raw data every day, and this number is estimated to double approximately every 40 months (McAfee and Brynjolfsson, 2012). By 2020, we are expected to exceed the threshold of 40 zettabytes of information (Mellor, 2015). The main reasons for this data explosion lie in the technological advances in many fields of the IT industry resulted in easily available, affordable and, as a result, ubiquitous hardware resources. Six of the world's 7 billion people own at least one mobile phone, embedded microprocessors have penetrated almost every aspect of humans' life, networking technologies and the

[1] 1 TB = 10^{12} bytes

Internet made it possible for smart devices to be connected and accessed remotely – these are just a few examples of factors contributing to the exponential growth of created datasets.

Velocity: on-the-fly processing of streaming data

In addition to the extreme size of created data, the ubiquitous insertion of mobile and embedded devices also affected the rate at which it is generated. Mobile phones, sensor networks, social networks, embedded systems – all these emit data every single second and thus contribute to the velocity aspect of Big Data. Even though the Big Data revolution started from the volume aspect, more and more applications focus on the speed of data creation and processing, seeing it as a key to success in taking timely business decisions.

Variety: heterogeneous data is stored in many different forms and formats

Relational databases are not the only one way of storing data. Moreover, as datasets grow in their size, the relational format discontinues being the primary form of data storage. Indeed, in the era of Internet and social media, data takes the form of video and audio clips, images, and texts. YouTube, for example, reports 300 hours of video being uploaded by users every minute (YouTube, LLC, 2015), and this rate constantly grows. Similar things happen with Facebook, Twitter, Google+ and other social media. Another contributing factor to the variety aspect is the presence of numerous vendors and providers, and lack of standardisation in many areas – for example, different sensor devices use different, often incompatible, data formats. Even the difference in human languages, used to express simple text data, creates additional heterogeneity among distinct datasets.

Veracity: data is often uncertain, flawed, or rapidly changing

With current rates of data creation, it is becoming more and more difficult to know which information is still accurate and valid, and which one is obsolete and outdated. For instance, reacting to stock exchange fluctuations even with a one-minute delay may lead to dramatic money losses. Same applies to, for example, environmental sensor readings and Global Positioning System (GPS) signals, whereas other data is less sensitive to timely, real-time reacting, but still have a 'date of expiry' – a time limit, within which analysis results correctly reflect the current situation and, therefore, are still valid. Trust is another dimension of the veracity aspect, becoming particularly important with the development of the so-called 'wisdom of the crowds'. With the proliferation of social networks and Web

sites, such as Wikipedia,[1] Yahoo! Answers,[2] or Quora,[3] where information heavily depends on subjective, often biased human opinion, it can be expected to be less trusted source of information.

Even when considered separately from the rest, each of the four aspects of Big Data on its own represents a pressing challenge and deserves thorough investigations so as to create efficient solutions. When taken together, they become an even more challenging task to be addressed both by the industry and the academia, and demand for novel technological solutions to be applied in this respect.

3.1.1 Processing streaming Big Data

Even though technological advances in handling static data – or 'data at rest' – seem to be coping with the quantity of data (e.g., the MapReduce computing model (Dean and Ghemawat, 2008) or so-called NoSQL databases (Pokorny, 2013)), the problem may often appear artificial – it is not the vast amounts of static data which have to be stored and processed in novel ways, it is the lack of efficient, reliable and optimised solutions for processing data in motion, which forces enterprises to permanently store their data on hard drive first and then process them in a static manner (Wähner, 2015). Indeed, if we take a more precise look at the data volumes generated by the world today, we will see that big data sources feed data unceasingly in real time and the majority of the data is streaming by its very nature – social network updates, stock exchange fluctuations, sensor readings, purchase transaction records, mobile phone and GPS signals, live video and audio content, etc. An increasing number of distributed applications are required to process continuously streamed data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries.

Raw data generated on a daily basis comes from everywhere – sensors networks, posts to social media sites, digital pictures and videos, purchase transaction records, stock exchange fluctuations, to name a few. Even though existing technologies seem to succeed in storing these overwhelming amounts of data, on-the-fly processing of newly generated data is an inherently pressing task. If tackled by the traditional DBMSs, the task of processing continuously streamed data from geographically distributed sources at unpredictable rates to obtain timely responses to complex queries, will be hindered by two main factors (Margara and Cugola, 2011). Firstly, in relational databases data is supposed to be (persistently) stored and indexed before it can be processed. Secondly, data is typically pro-

[1] https://en.wikipedia.org/
[2] https://answers.yahoo.com/
[3] https://www.quora.com/

cessed only when explicitly queried by users (i.e., asynchronously with respect to its arrival).

Information Flow Processing (IFP) – a key research area addressing the issues involved in processing streamed data – investigates potential solutions addressing these limitations of the traditional static approaches. IFP focuses on data flow processing and timely reaction (Margara and Cugola, 2011). The former assumes that data is not persisted, but rather continuously flowing and being processed in memory, and the latter means that IFP systems aim to operate in real-time mode, and time constraints are crucial for them. These two key features have led to the emergence of a family of computer systems specifically designed to process incoming data streams based on a set of pre-deployed processing rules. A data stream consists of an unbounded sequence of values continuously appended and annotated with a timestamp, usually indicating when it has been generated (Calbimonte et al., 2012). Timestamps allow for stream processing solutions then to order incoming tuples in a chronological order. Usually (but not necessarily) recent tuples are more relevant and useful, because they represent a more up-to-date situation, and therefore are more helpful in achieving near real-time operation. Examples of data streams include environmental sensor readings, stock market tickers, social media updates, etc.

Querying over data streams

To cope with the unbounded nature of streams and enable data processing, so-called continuous query languages (Calbimonte et al., 2012) have been developed to extend the conventional Structured Query Language (SQL) semantics with the notion of *windows*. A window is a temporal operator, which uses tuple timestamps to transform unbounded sequences of values into bounded ones, allowing the traditional relational operators to be then applied to the resulting collection of tuples. This approach restricts querying to a specific window of concern, which consists of a subset of most recent tuples, while older values are (usually) ignored (Barbieri, Braga, Ceri, Della Valle and Grossniklaus, 2010). Windows can be specified in terms of:

- **number of elements (tuples)**, when a window consists of a number of latest elements regardless of the arrival time, and
- **time**, when a window consists of all elements which have arrived during the specified time frame[1] (in this case, the window can be potentially empty).

[1]This division into tuple- and time-based windows is also referred to as physical and logical extraction respectively (Barbieri et al., 2009).

Depending on how the window operator 'moves' along the data stream, we can distinguish between overlapping and non-overlapping windows. With the former approach (also known as sliding), the transition between windows is smooth, such that two neighbour windows may overlap with each other and same tuples may appear in both of them. In the latter case, also known as tumbling, the transition between windows is 'discrete', so that a tuple can appear only in at most one window.

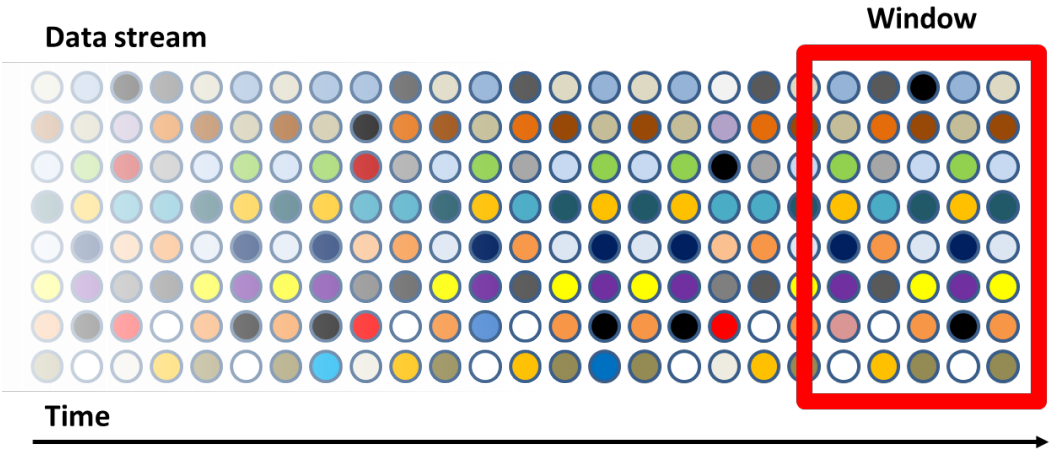

Figure 3.1: Continuous query languages address the problem of querying an unbounded data stream by focussing on a well-defined window of interest (excerpted from (Dautov et al., 2014*b*)).

The concepts of unbounded data streams and windows are visualised in Figure 3.1. The multi-coloured circles represent tuples continuously arriving over time and constituting a data stream, whereas the thick rectangular frame illustrates the window operator applied to this unbounded sequence of tuples. As time passes and new values are appended to the data stream, old values 'fade away' – they are pushed out of the specified window, i.e., become no more relevant and may be discarded (unless there is a need for storing historical data for later analysis).

Data stream management systems (DSMSs) – an evolution from traditional static DBMSs – were specifically designed and developed to process streaming data, which comes from different sources to produce new data streams as output (Margara and Cugola, 2011). Examples of DSMSs include SQLstream,[1] STREAM,[2] Aurora,[3] TelegraphCQ,[4] etc. These systems target transient, continuously updated data and run standing (i.e., continuous) queries, which fetch updated results as

[1] http://www.sqlstream.com/
[2] http://infolab.stanford.edu/stream/
[3] http://cs.brown.edu/research/aurora/
[4] http://telegraph.cs.berkeley.edu/telegraphcq/v0.2/

new data arrives. CEP goes beyond simple data querying aims to detect complex event patterns, themselves consisting of simpler atomic events, within a data stream (Margara and Cugola, 2011). Accordingly, from CEP's point of view, constantly arriving tuples can be seen as notifications of events happening in the external world – e.g., a fire alarm signal, social status update, a stock exchange update, etc. Accordingly, the focus of this perspective is on detecting occurrences of particular patterns of (lower-level) events that represent higher-level events. A standing query fetches results (i.e., notification of a complex event to the interested parties is sent) if and only if a corresponding pattern of lower-level events is detected. For example, a common task addressed by CEP systems is detecting situation patterns, where one atomic event happened after another. To achieve this functionality, CEP systems also rely on tuple timestamps; they extend continuous query languages with sequential operators, which allow specifying the chronological order of tuples or, simply put, whether one tuple arrives before or after another in time.

3.1.2 An existing solution: IBM InfoSphere Streams

There are several technologies, both commercial (e.g., IBM InfoSphere Streams (IBM Corporation, 2013) and Oracle Fast Data solutions (Oracle Corporation, 2015)) and open-source (e.g., Apache S4[1] or Storm[2]), which were specifically developed to handle streaming data and process it in memory, without permanently storing it on disk. They are designed to supports dynamic analysis of massive volumes of incoming data to improve the speed of business insight and decision making. Typical features include:

- A wide selection of pre-compiled operators, ranging from simple utility operators (e.g., source, sink, filter, aggregate, split) to more complex ones, such as Extensible Markup Language (XML) Parser or operators dealing with database access.
- Support for task parallelisation: with various built-in operators responsible for splitting, filtering and merging incoming data streams it is possible to achieve highly parallelised behaviour of a Streams application.
- Support for scalability: configuring and provisioning of additional processing components is possible to support parallel execution of data processing tasks.

[1] `http://incubator.apache.org/s4/`
[2] `http://storm.apache.org/`

In the context of the presented research work we have employed IBM InfoSphere Streams (henceforth – IBM Streams). Streams consists of a programming language Streams Programming Language (SPL), an IDE, and a runtime environment that can execute SPL applications in stand-alone or distributed modes. The Streams Studio IDE includes tools for creating visual representations of applications, consisting of operators and connecting streams, by means of the drag-and-drop mechanism. Alternatively, users can also develop Streams applications directly in SPL. The following are the main advantages of Streams:

- Apart from standard operators, Streams also supports custom operators, which can be written in SPL, Java or C++. Thanks to this feature, it is possible to package the EXCLAIM framework as a separate Java Archive (JAR) file and access it as an operator from within a Streams application.

- The feature-rich and user-friendly Streams IDE is based on Eclipse IDE, which makes it intuitively easy to use for someone with experience of developing in Eclipse. In Streams IDE, provisioning of an additional analysis component is just a matter of dragging and dropping it on the working area.

- Automated 'zoo keeping' is another positive feature of the Streams platform. While in industrial and commercial scenarios, one might want to have more control and visibility into how distributed computational nodes are coordinated and managed, in our work we were looking for an easiest and quickest way of running parallel task without much upfront configuration effort. In the Streams platform, this process of configuring and managing computational nodes known as 'zoo keeping') is done automatically and transparently to the user. Whereas in open-source solutions, such as Apache Storm or S4, this process has to be done manually.

- The last but not the least, being a commercial software offering, IBM Streams is also available as a free Linux distribution for non-commercial research use. This distribution has exactly the same set of features as the full-blown commercial offering with the only difference that a distributed cluster of nodes is simulated within a single virtual machine, as opposed to a physical cluster in a data centre or in a cloud. Therefore, with certain assumptions it can claimed that local experiments can be reproduced on a larger scale.

However, IBM Streams and other platforms for processing large amounts of streaming data in a parallel manner should not be considered as a 'silver bullet', which will immediately solve all the problems associated with Big Data processing. They only provide an environment and a set of tools for implementing these

tasks. It is up to the users, who are expected to design and develop correct fragmentation and processing algorithms for incoming data. We describe our own experience of utlising IBM Streams in Chapter 7.

3.2 Semantic Sensor Web

One of the main problem domains, where solutions based on the principles of stream processing have been successfully applied, is the domain of distributed physical sensor networks, which also act as one of the main driving factors for development of the stream processing technology. Sensor networks are spatially distributed autonomous sensors, whose main goal is to monitor physical or environmental conditions, such as temperature, humidity, sound, pressure, etc., and to pass this monitored data through the wireless (or less often wired) network to one or several central component responsible for further information processing and storage. A sensor network (see Figure 3.2) includes multiple nodes, which are either i) sensor nodes – physical devices carrying one or more sensors, or ii) gateway nodes for transferring data to the central processing and storage location. Each sensor node is typically equipped with an interface for transferring data over the network (e.g., a radio transceiver with an internal antenna), a microcontroller and an energy source (e.g., an embedded battery or a solar panel). Nodes and sensors may vary in size, complexity and price. A sensor network consists of multiple nodes ranging from a few (e.g., a body-area network for monitoring human's health) to several hundreds and even thousands (e.g., an air pollution monitoring system) assets.

Until recently, sensor networks could be regarded as relatively scattered and isolated groups of sensor components, each based on its own proprietary standards and closed architectures, and serving its own individual purposes (Dautov et al., 2014*a*). Often, they were not discoverable and remotely accessible via the Internet and the collected data was only accessible by the owner of a particular network. In these circumstances, the usage of the sensor technology was limited. Firstly, the access to the network and its individual nodes was only possible with the physical presence of an operator – that is, if something was wrong, the operator had to go and manually inspect a malfunctioning node. Secondly, the proprietary standards and closed architectures made it impossible for sensors and sensory data to be accessed and reused by third parties. It often resulted in situations where multiple sensors of a similar type were installed in the same physical location just because the already installed sensors could not be accessed by newcomers.

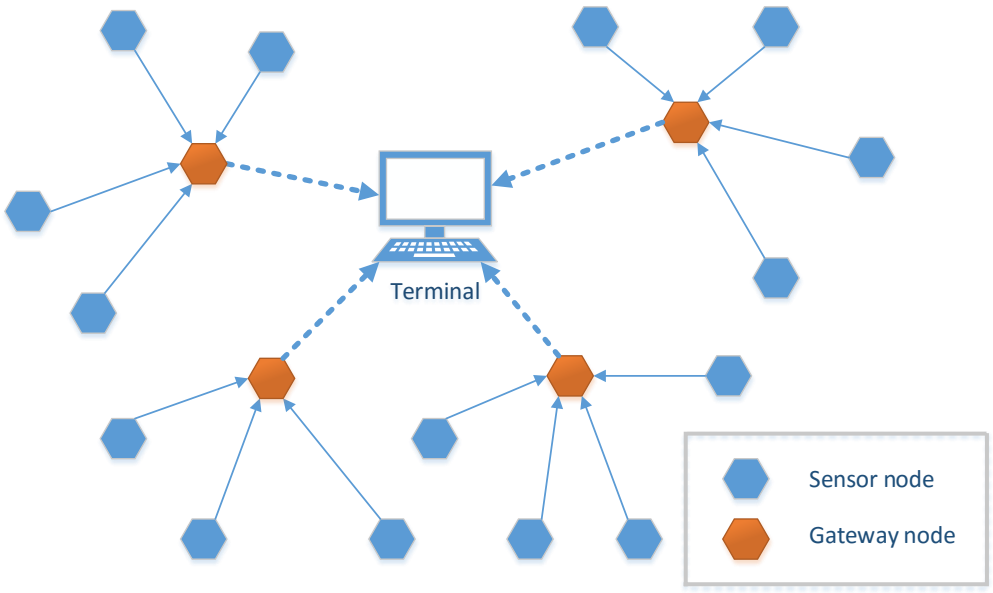

Figure 3.2: A sensor network consists of multiple sensor nodes and several gateway nodes, which transfer data to the central terminal (modified from (Dautov et al., 2014*a*)).

This situation started changing in the beginning of the 2000s with the launch of active collaborative efforts by the Open Geospatial Consortium (OGC).[1] OGC is an international industry consortium of more than 500 companies, government agencies and academic institutions, who actively collaborate to develop publicly available interface standards and their implementations, related to geospatial content and services, Geographic Information Systems (GISs), and associated data processing and data sharing (Lupp, 2008). Standards developed by OGC aim at bringing the spatial dimension to the Web, wireless and location-based services and mainstream IT in general. The OGC standards enable technology developers to design and create complex 'geo-enabled' information services, accessible and usable with a wide range of hardware and software assets.

Accordingly, one of the main directions for the OGC standardisation collaboration is the domain of sensor networks, which has become known as the SWE[2] initiative (Botts et al., 2008). SWE sets as its main goal the connection of scattered and isolated sensor networks into a globally-accessed and open ecosystem existing on top of the World Wide Web. The consortium members are specifying interoperability interfaces and meta-data encodings that enable real-time integra-

[1] http://www.opengeospatial.org/
[2] http://www.ogcnetwork.net/\gls{swe}/

tion of heterogeneous sensor networks into the single information infrastructure – the Sensor Web. To fulfill this global vision, SWE develops standard encodings and web services, to enable (Open Geospatial Consortium, Inc., 2015):

- online discovery of sensors, processes, and sensory data
- remote access to and tasking of sensors
- remote access to sensory static and streaming data
- publish-subscribe capabilities to enable all kinds of notifications
- robust sensor systems and process descriptions

Accordingly, the Sensor Web can be seen as a collection of protocols and API, coupled to and providing access to an interconnected network of Web-accessible sensor networks and historical sensory data repositories.

From the Sensing Cloud to the Semantic Sensor Web

The concept of the Sensor Web goes hand in hand with the development of the Internet of Things (IoT) (Atzori et al., 2010). The IoT is a relatively novel concept whose basic underlying idea is the ubiquitous presence of a variety of digital objects – i.e., things – connected to the Web and remotely accessible via standard protocols and interfaces. Examples of such objects include Radio Frequency Identification (RFID) tags, various sensors and actuators, mobile phones, cameras, etc.

The SWE community sets as its goal the creation of a similar architecture, which has become known as the Sensing Cloud (Distefano et al., 2015) – a pervasive network of sensor devices, each connected to the Web and remotely accessible by machines and humans. The correct implementation of the Sensing Cloud initiative allows real-time access and browsing of the physical world through millions of sensing devices, and thereby facilitates all kinds of situation assessment tasks. Each sensor, constituting the Sensing Cloud, is uniquely identified with its Universal Resource Identifier (URI) and accessed by means of standard network protocols. Accordingly, its monitored data can be accessed as a simple Web page or continuously streamed to create more complex real-time monitoring applications.

Apart from data privacy and security, which have been always identified among the key factors preventing the implementation of the SWE and the Sensing Cloud (Atzori et al., 2010), there are also other impediments. The number of sensors around the globe is constantly growing, which results in avalanches of generated sensory data. Such a rapid development of the sensor technology is intensifying

the existing problem of too much data and not enough knowledge. Accordingly, one of the main challenges, which the SWE community has to face, is the existing heterogeneity in sensory data models, formats and representations. With the introduction of standard formal schemes for describing individual sensors and sensor networks, as well as associated sensory data models, the situation improved. However, these standards, mainly syntactic, XML-based (see, for example, the SensorML specification (Botts and Robin, 2014)), were suffering from low expressivity, insufficient interoperability and integration capabilities at the semantic level. It meant that despite the same high-level standard methodologies used to design various sensor networks and associated data models, the actual low-level implementation details might have been different. For example, different sensor networks might use different units of measurements or human languages to describe same concepts.

This situation naturally called for application of semantic technologies, which would allow sensory data to be annotated with semantic metadata with a goal to increase interoperability between distinct heterogeneous sensor networks, as well as to provide contextual information essential for situation awareness. The application of semantic technologies is based on a simple yet efficient principle – by mapping between different syntactic metadata schemas in a structured way, the Semantic Web techniques can tackle the problem of data integration and discovery in the context of the Sensor Web.

This is where the expressive power and support for formal reasoning of the Semantic Web technologies, and more specifically – ontologies, have proved to be useful. Ontologies and other semantic technologies enabled sensor networks with higher semantic interoperability and integration capabilities, as well as facilitated reasoning, classification and other types of automated assurance. Accordingly, the marriage of the Sensor Web with the Semantic Web technologies has become known as the SSW (Sheth et al., 2008). The main idea behind the SSW is the annotation of sensor descriptions and sensor observations with Semantic Web languages. These annotations serve to provide more meaningful and expressive descriptions, advanced access and formal analysis of sensor data than SWE alone, and act as a linking mechanism to bridge the gap between the primarily syntactic XML-based metadata standards of the SWE and the RDF/OWL-based metadata standards of the Semantic Web.

By carefully following the SWE standardisation guidelines for modelling sensor networks, the SSW annotates sensory data with spatial, temporal, and thematic ontological metadata. A semantically-enhanced sensor network or simply – an *SSN* – allows the network, its sensor nodes and associated sensory data to

be organised, installed and managed, queried, understood and controlled through high-level specifications (Compton et al., 2012). Ontologies for the sensor domain provide a vocabulary for describing sensors, and allow classification of the capabilities and measurements of sensors, provenance of measurements, and besides that, also enable formal reasoning on sensor resources (Compton et al., 2009). As a result, semantic annotations enable interoperability by connecting formerly scattered and heterogeneous sensor resources into a common infrastructure, while ontologies and rules serve to support analysis and reasoning over sensor data in the SSW (Sheth et al., 2008).

At the core of the SSW approach lies the SSN ontology – an extensible semantic vocabulary which aims at describing sensor-enabled domains in an abstract, yet comprehensive manner. Before explaining the SSN ontology in more details, we first brief the reader on the Semantic Web and its main enabling technologies to help better understand the rest of this chapter, as well as the material to be explained in Chapters 5 and 6.

3.2.1 Semantic Web and the Semantic Web stack

The Semantic Web, introduced by Berners-Lee (Berners-Lee et al., 2001) in 2001, is the extension of the World Wide Web that enables people to share content beyond the boundaries of applications and websites (Hitzler et al., 2009). This is typically achieved through the inclusion of semantic content in web pages, which thereby converts the existing Web, dominated by unstructured and semi-structured documents, into a web of meaningful machine-readable information. Accordingly, the Semantic Web can be seen as a giant mesh of information linked up in such a way as to be easily readable by machines, on a global scale. It can be understood as an efficient way of representing data on the World Wide Web, or as a globally linked database.

As shown in Figure 3.3, the Semantic Web is realised through the combination of certain key technologies (Hitzler et al., 2009). These technologies from the bottom of the stack up to the level of XML have been part of the Web standardised technology stack even before the emergence of the Semantic Web, whereas the upper, relatively new technologies – i.e., Terse RDF Triple Language (Turtle) and Notation3 (N3), RDF, RDF Schema (RDFS), SPARQL Protocol and RDF Query Language (SPARQL), OWL, and SWRL – are intrinsic to the Semantic Web research. All of these components have already been standardised by the World Wide Web Consortium (W3C) and are widely applied in the development of Semantic Web applications. Briefly, these standards and technologies are:

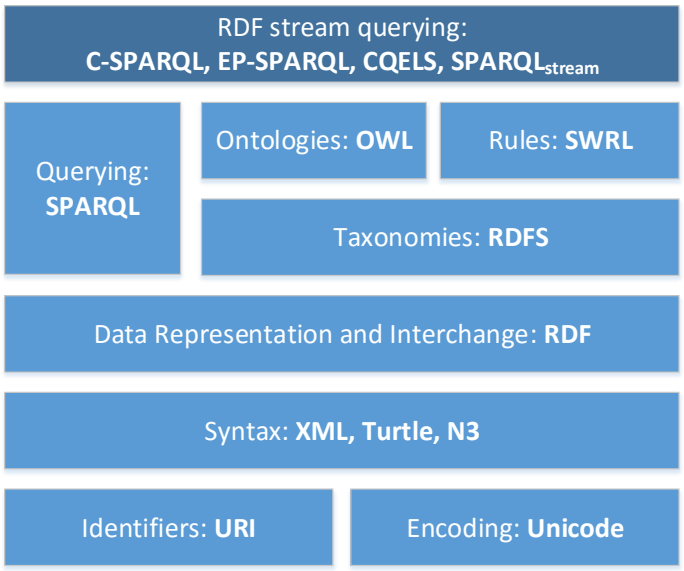

Figure 3.3: Semantic Web stack.

- **RDF** serves to represent information in the form of so-called 'triples', each of which consists of a subject, a predicate, and an object, expressed as a Web URI. RDF enables the representation of information about resources in the form of graph – that is why the Semantic Web is sometimes called a Giant Global Graph. As noted above, an RDF-based data model can be represented in a variety of syntaxes (e.g., RDF/XML, N3, and Turtle). For example, the following listing contains RDF serialisations for a simple statement, which can be expressed with a natural language as 'Sheffield is a city located in England'. Accordingly, Listing 3.1 contains the RDF/XML serialisation, and Listing 3.2 contains the N3/Turtle notation.

Listing 3.1: Example of the RDF/XML serialisation.

```
<rdf:\gls{rdf}
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:pl="http://purl.org/ontology/places#">
<rdf:Description rdf:about="http://en.wikipedia.org/wiki/Sheffield">
<rdf:type>pl:City</rdf:type>
<pl:hasLocation>http://en.wikipedia.org/wiki/England<pl:hasLocation>
</rdf:Description>
</rdf:\gls{rdf}>
```

Listing 3.2: Example of the N3/Turtle serialisation.

```
@prefix rdf: http://www.w3.org/1999/02/22-rdf-syntax-ns.
```

```
@prefix pl: http://purl.org/ontology/places.
<http://en.wikipedia.org/wiki/Sheffield>
rdf:type pl:City;
pl:hasLocation https://en.wikipedia.org/wiki/England.
```

In our work in general and in this document, we have adopted the Turtle notation for its relative compactness, simplicity and human-readability.

- **RDFS** provides a basic schema language for RDF. For example, using RDFS it is possible to create hierarchies of classes and properties – that is to create simple, light-weight taxonomies, also known as RDF vocabularies, intended to structure RDF resources. These resources can be saved in a special non-relational database, called a triple store, and further queried with the special query language SPARQL.

- **SPARQL** is an RDF query language, which serves to query any RDF-based data from triple stores, including statements involving RDFS and OWL. Its syntax, mainly inspired by the established SQL, allows to define queries consisting of triple patterns, conjunctions, disjunctions, and optional patterns. SPARQL is considered to be one of the key technologies constituting the Semantic Web stack, and to date, there exist multiple implementations of it, written in many programming language. The following sample human-readable query in Listing 3.3 demonstrates the syntax of SPARQL. The query fetches all cities, located in England (e.g., Sheffield).

Listing 3.3: Example of a SPARQL query.

```
//define a shortcut prefix to save space and make
//the code more human-readable
PREFIX pl: http://purl.org/ontology/places
//query all cities from the triple repository...
SELECT ?city
//...which are located in England
WHERE {
?city a pl:City.
?city pl:hasLocation https://en.wikipedia.org/wiki/England.
}
```

- **OWL** is a family of knowledge representation languages used to formally define an ontology – "a formal, explicit specification of a shared conceptualisation" (Studer et al., 1998). Typically, an ontology is seen as a combination of a terminology component (i.e., TBox) and an assertion component (i.e., ABox), which are used to describe two different types of statements in ontologies. The TBox contains definitions of classes and properties, whereas

the ABox contains definitions of instances of those classes. Together, the TBox and the ABox constitute the knowledge base of an ontology. OWL extends RDFS by adding more advanced constructs to describe resources on the Semantic Web. By means of OWL it is possible to explicitly and formally define knowledge (i.e., concepts, relations, properties, instances, etc.) and basic rules in order to reason about this knowledge. OWL allows stating additional constraints, such as cardinality, restrictions of values, or characteristics of properties such as transitivity. The OWL languages are characterised by formal semantics – they are based on *Description Logics* (DLs) and thus bring reasoning power to the Semantic Web. There exists a prominent visual editor for designing OWL ontologies, called Protege,[1] and several reasoners written in multiple programming languages, such as Pellet,[2] FaCT++,[3] and HermiT.[4]

- **SWRL** extends OWL with even more expressivity, as it allows defining rules in the form of implication between an antecedent (body) and consequent (head). It means that whenever the conditions specified in the body of a rule hold, then the conditions specified in the head must also hold. It is worth noting, that fully compatible with OWL-DL, SWRL syntax is quite expressive, which may have certain negative impacts on its decidability and computability. Listing 3.4 contains a rule, expressed in a human-readable syntax, and illustrates the functionality of SWRL. The following sample states that if a city is located in England, then it is also located in the United Kingdom.

Listing 3.4: Example of a SWRL rule.

```
pl:City(?city) ^ pl:hasLocation(?city, "http://en.wikipedia.org/wiki
    /England") -> pl:hasLocation(?city, "https://en.wikipedia.org/
    wiki/United_Kingdom")
```

Note that with OWL and SWRL, there is typically more than one way of defining knowledge deducing facts. For example, this very same inference achieved by reasoning over the SWRL rule, can be also achieved by defining `hasLocation` as a transitive property between a city and England, and also between England and the United Kingdom. Even though it is not explicitly stated that a city is located in the United Kingdom, the reasoner will deduce this based on the fact that England is located in the United Kingdom by following the transitive property.

[1] `http://protege.stanford.edu/`
[2] `https://github.com/complexible/pellet`
[3] `https://code.google.com/p/factplusplus/`
[4] `http://hermit-reasoner.com/`

3.2.2 SSN ontology

The collaborative efforts on applying semantic technologies to the Sensor Web, supported by the W3C consortium and its SSN Incubator Group,[1] have resulted in the development of the Semantic Sensor Networks (SSN) ontology – the core ontological vocabulary for annotating sensor resources with semantic descriptions so as to address the requirements of the SSW. The SSN ontology, consisting of 41 concepts and 39 object properties (Compton et al., 2012), can be used to model individual sensor devices and whole networks, their environments, processes, and sensor observations. The SSN ontology is encoded with OWL and is receiving more and more attention even beyond the context of the Sensor Web research (e.g., in (Schlenoff et al., 2013) authors explore how it can be extended to be used in the manufacturing domain). Figure 3.4 provides a bird's-eye view on the SSN ontology and its main components.

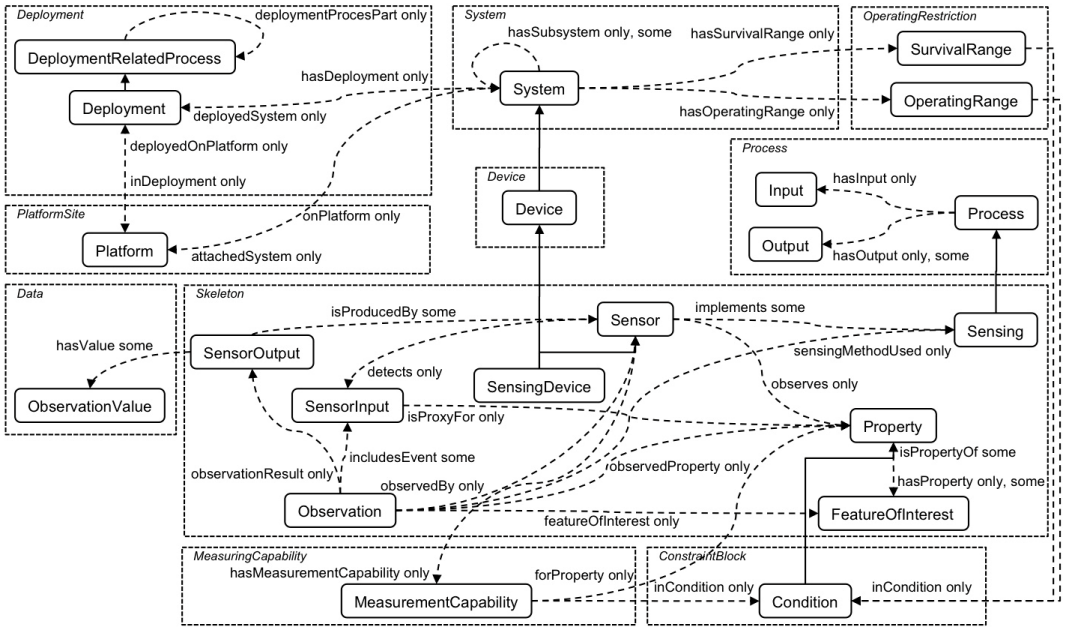

Figure 3.4: A bird's-eye view on the SSN ontology (excerpted from (Compton et al., 2012).

As it is seen from the diagram, the SSN ontology provides a generic high-level framework and vocabulary of terms, covering different aspects of various sensor-enabled scenarios, from actual sensor descriptions and their measuring capabilities to the deployment context and surrounding environment. Following the principles of the Semantic Web and ontology engineering, it is explicitly designed with an intention to be extended with low-level concepts to meet requirements of

[1] http://www.w3.org/2005/Incubator/ssn/

specific sensor-enabled domains.

Special attention is also paid to modelling sensory data and observations. With the support of the SSN ontology and its extensions, it is possible to create a semantic extra-layer on top of the existing syntactic sensory data and map between heterogeneous sensory data formats and encodings. It means that sensor devices can either i) send their observations in the form of RDF triples constructed from the SSN classes and properties, or ii) embed semantic metadata into existing syntactic data (e.g., semantic annotations within XML tags). It is also possible to introduce certain hardware and software components responsible for data transformation and mapping between the source format and SSN-enabled semantic annotations. On the one hand, the integration of formerly-heterogeneous sensory data streams by means of a common ontological vocabulary opens wide opportunities for increased sensor interoperability and sensory data processing and analysis. On the other hand, it requires additional efforts to be put into investigation of how such semantically annotated RDF data streams should be properly queried and processed in real time. The research area, which looks into this intersection of stream processing and the Semantic Web, is known as the RDF Stream Processing.

3.2.3 RDF stream processing

The SWE aims at connecting existing heterogeneous sensors into a global Web-based network. Together with other data streams becoming more and more common on the Web (e.g., stock exchange movements, social network updates, etc.), the application of stream processing techniques to avalanches of streaming data generated within the Web came as a natural fit. Among other challenges, this required novel ways of coping with the typical openness and heterogeneity of the Web environment – in this context, the role of the Semantic Web languages has become to facilitate data integration an 'homogenisation' in open environments, and thus help to overcome these problems by using uniform machine-readable descriptions to resolve heterogeneities across multiple data streams (Lanzanasto et al., 2012). This in turn led to the emergence of the SSW, which imposes encoding of sensor data in the form RDF triples and, accordingly, opens new opportunities for performing dynamic reasoning tasks over RDF data streams to support real-time situation assessment. RDF stream processing[1] research goes beyond the

[1]Originally, this research domain was known as Stream Reasoning, which was first coined and promoted by Della Valle et al. (Della Valle et al., 2009). The term, however, is misleading – to date, existing solutions for RDF stream querying do not provide support for formal reasoning to the same extent as the existing querying technologies (i.e., SPARQL) for traditional, static RDF querying. For this reason, in the rest of this document we will be using the term RDF stream processing, which reflects the actual state of the art more precisely and accurately.

existing stream processing and CEP approaches by aiming at enhancing the continuous querying with the run-time reasoning support – that is, with capabilities to infer additional, implicit knowledge based on already given, explicitly stated facts. Barbieri et al. (Barbieri, Braga, Ceri, Valle, Huang, Tresp, Rettinger and Wermser, 2010) define it as "reasoning in real time on huge and possibly noisy data streams, to support a large number of concurrent decision processes".

As Semantic Web technologies are mainly based on DLs (Hitzler et al., 2009), their application to data stream processing in turn also enabled formal reasoning tasks over continuously and rapidly changing information flows. To date, there exist several prominent RDF stream processing approaches developed by the SSW community – C-SPARQL (Barbieri et al., 2009), CQELS (Le-Phuoc et al., 2011), EP-SPARQL (Anicic et al., 2010), and SPARQLstream (Calbimonte et al., 2012) to name a few.[1] All of them extend the conventional SPARQL syntax with a specific support for handling data streams, consisting of RDF triples. These continuous RDF query languages aim at preserving the core value of stream processing, i.e., querying streamed data in a timely, in-memory fashion, while providing the following additional benefits (Lanzanasto et al., 2012):

- **Support for formal reasoning**: depending on the extent to which stream reasoning systems support reasoning, it is possible not only to detect patterns of events (as CEP already does), but additionally to perform more sophisticated and intelligent event detection by inferring implicit knowledge based on pre-defined facts and rules (i.e., static background knowledge).

- **Integration of streamed data with static background knowledge**: with RDF stream processing, it is possible to match streaming tuples against a static background knowledge base (usually represented as an RDF data set). This modular separation of concerns allows, for example, performing run-time, dynamic data analysis and situation assessment with respect to previously recorded historical observations.

- **Support for expressive queries and complex schemas**: ontologies (e.g., the SSN ontology) serve as common vocabularies of terms for defining complex expressive queries, where ontological classes and properties provide 'building blocks' and may be used for defining standing queries.

- **Support for temporal and sequential operators**: to cope with the unlimited continuous nature of data streams, RDF stream processing systems extend

[1] We refer interested users to (Margara et al., 2014) for a detailed survey of existing RDF stream processing approaches.

established SPARQL logical and data operators to enable limiting an unbounded stream of RDF triples to a specific window, and also to detect RDF events, which follow one after another chronologically.

In Figure 3.3, RDF stream processing is located at the very top of the stack. The existing technologies are not yet standardised, but initial steps in this direction have already been taken – W3C RDF Stream Processing Community Group[1] collaborates on integrating existing heterogeneous technologies into a common standardised approach.

3.3 Summary

The chapter familiarised the reader with Big Data processing – a novel and still developing research area, which focuses on enabling scalable storage and processing of large heterogeneous datasets. These capabilities are required to support timely data monitoring and analysis in cloud platforms. The chapter first explained the notion of Big Data and its four main aspects, known as the 'four Vs' – namely, Volume, Velocity, Variety and Veracity. When talking about processing, we explicitly distinguished between static and streaming approaches. The latter enable dynamic in-memory data processing and address the Velocity aspect of Big Data.

However, often there is also a requirement to address the Variety aspect, which is beyond the capabilities of traditional stream processing approaches targeting at purely syntactic data. To explain how data heterogeneity can be addressed, the second part of the chapter exposed the reader to the SSW research. This relatively novel research area is a combination of the Sensor Web with the Semantic Web technologies, and stems from the requirement to bridge the gap between various heterogeneous, primarily syntactic data formats for representing sensor resources. To enable uniform data representation, advanced data access, and increased interoperability, the SWE consortium employed and successfully applied existing Semantic Web technologies. Main contribution of the resulting marriage, which has become known as the SSW, are the SSN ontology – the primary vocabulary for modelling sensors, sensor networks, data and other sensor-related resources – and several approaches for run-time dynamic querying of streaming RDF data in the context of SSW applications. As explained in Chapter 5, based on our interpretation of service-based cloud platforms as networks of distributed software sensors, we apply existing solutions from the SSW research – namely, the SSN ontology and RDF stream processing – to the domain of data monitoring and analysis with a goal to overcome heterogeneity existing among individual sensors.

[1] https://www.w3.org/community/rsp/

Part II

State of the art and our approach to address the gaps

Part II of this document can be split into two sub-parts. First, it surveys the current state of the art in the relevant research area. The goal of this literature survey is two-fold. Firstly, it is important to understand common established practices and, potentially, re-use and build on the existing work. Secondly, this literature survey serves to identify existing research and technological gaps in the considered research domain, and accordingly position our own research work with respect to the rest of the research work. Outlined gaps are useful in outlining preliminary high-level functional properties for the future EXCLAIM framework, which is described in details in the second sub-part.

Second, this part is dedicated to the actual description of our approach to performing monitoring and analysis activities in the context of heterogeneous service-based CAPs. It relies on the material presented so far – namely, background knowledge, related technologies and the state of the art survey. This part of the thesis includes three chapters. Accordingly, in Chapter 5 we first present a high-level conceptual overview of the EXCLAIM framework. Here, we present and explain the fundamental underpinnings, which provide a basis for designing and implementing the framework. Chapter 6 builds upon and further extends the material presented in Chapter 5 with implementation details of the prototype version of the framework. It also presents the Cloud Sensor Ontology (CSO) as the core component of the framework, which is utilised in every step of the monitoring and analysis process. Chapter 7 presents a case study, which serves to conduct experimental evaluation of the framework. The experiments are intended to demonstrate performance and scalability of the framework.

Chapter 4

State of the art in cloud platform self-governance

The goal of this chapter is two-fold. We first introduce the state of the art in the domain of data monitoring and analysis in service-based cloud environments, systematically classifying the overall body of research work, and providing the reader with an understanding of existing techniques, tools and approaches. We then use this classification to identify technological gaps in this existing research body and position our own work with respectively. Accordingly, the identified gaps will serve to outline fundamental requirements for the EXCLAIM framework and will help to evaluate the framework's benefits with respect to other approaches.

4.1 Overview of the IaaS and SaaS self-management

To date, advances in enabling clouds with autonomic behaviour have mainly been made at the IaaS level[1]. Building on techniques developed by the grid and High Performance Computing (HPC) computing communities, to support elasticity and load-balancing, these existing approaches usually rely on collecting and interpreting such data as CPU load, memory utilisation and network bandwidth to execute necessary adaptation actions. Depending on diagnosis and adaptation policies, such actions target various goals (e.g., increasing application performance, reducing cost and optimising energy consumption) and typically include replication and resizing techniques (Galante and de Bona, 2012). The former refers to adding and removing computational instances to meet ever-changing resource requirements, whereas in the latter case resources are added or removed within a single compu-

[1]For a survey of existing elasticity and load balancing techniques, the interested reader is referred to (Galante and de Bona, 2012).

tational instance. Load balancing techniques are often used in combination with replication to spread workload evenly across available instances. Existing IaaS self-adaptation solutions can be further classified into reactive or predictive approaches. The former are the most widely adopted solutions for resource management in clouds and are commonly employed by many IaaS cloud providers. Their main principle is to react to critical situations only after they have been detected – that is, the monitoring and analysis mechanism aims at detecting patterns representing an already-happened event. The latter approaches use various heuristics and machine learning techniques to predict workload and scale resources accordingly so as to prevent a potentially critical situation from happening. In other words, the monitoring and analysis mechanism aims at detecting observational patterns which only represent 'symptoms', not the dangerous situation *per se*.

However, monitoring and analysis mechanisms, which would support more sophisticated adaptation scenarios, such as modification of the actual structure and/or behaviour of a deployed cloud application at run-time, are much more difficult to automate, and at the moment are beyond the capabilities of common cloud platforms. As an example, consider a situation when hundreds of applications deployed on a cloud platform are using the platform's built-in notification service (e.g., for e-mail notifications). At some point, this service crashes, affecting the QoS of all the dependent applications and potentially the whole platform. A possible solution in such circumstances would be the detection of such a situation and suggesting a possible adaptation plan so as to switch to an external notification service automatically and transparently to the users. Unfortunately, at the moment there seem to be no self-management mechanisms of such a kind in the PaaS segment of cloud computing. Even though much effort has been put into the development of self-management mechanisms at the IaaS level, similar capabilities for services at the PaaS level are as yet immature and not well theorised.

Even though various approaches have been described as targeting the PaaS level, these typically do not act at the PaaS-level directly, but rely on lower level, IaaS-related adaptation actions (Galante and de Bona, 2012). For example, having detected high CPU and memory utilisation levels for a service-based cloud application, a rather simplistic adaptation mechanism will simply allocate more hardware resources, whereas the actual reason for this excessive resource consumption was a malfunctioning add-on service. Therefore, in such circumstances a more intelligent adaptation solution would be to fix the single malfunctioning service by, for example, reconfiguring or replacing it – that is, by monitoring and interpreting data belonging to the PaaS level and applying adaptation at the platform, rather than at the infrastructure level.

Neither researchers nor platform providers have offered such solutions which would allow hosted applications to modify their internal structure and/or behaviour at run-time by adapting to changing context (e.g., by replacing one service with another). This task has instead been shifted to the SaaS level – that is, it has been left to software developers – the target customers of the PaaS offerings – to implement self-adaptation logic within specific applications. Application owners are expected to instrument their applications with monitoring and analysis capabilities so as to handle software interactions with platform add-on services and circumnavigate possible critical situations.

In these circumstances, it is our belief that with the rapid growth of the PaaS segment and wide adoption of CAPs, self-governance at this level of cloud computing is equally important, and development of self-adaptation mechanisms is essential in order to prevent service-based cloud platforms from dissolving into 'tangled' and unreliable environments. To some extent, self-governance in service-based cloud platforms is yet another aspect, which will enable them with an even richer support for software development and delivery of a reliable IoS. We see the PaaS segment as the place to implement self-governance in clouds – it has to build upon already existing solutions for IaaS self-governance, and, at the same time, exempt SaaS developers from implementing customised, application-specific logic in their software themselves. In our view, it has to provide a holistic and cross-layer solution to cloud self-management. In this light, with the presented research work, we aim to investigate how to create such an extensible mechanism for data monitoring and problem detection to support self-governance in service-based cloud platforms.

4.2 PaaS: State of the art in data monitoring and analysis in service-based cloud platforms

In this section we systematise the whole body of existing research efforts in the area of data monitoring and analysis of service-based cloud platforms with respect to a number of relevant and important criteria. These criteria include:

- *Data collection and monitoring* refers to how and from where monitored data is collected. Accordingly, surveyed approaches may differ in sources of monitored data (e.g., databases, text files, or JavaScript Object Notation (JSON) and XML messages) and level sof monitoring (i.e., infrastructure, application performance, service coordination, and business processes).

- *Data analysis* refers to the process of data interpretation after it was collected.

It defines to how timely (i.e., predictive, reactive, or 'post-mortem') and automated (i.e., manual, interactive, or automated) a particular analysis component is.

- *Framework architecture* refers to design and implementation aspects of surveyed approaches, and includes the following aspects: level of intrusiveness (i.e., intrusive or non-intrusive), component distribution (i.e., distributed or centralised architecture), language type (i.e., declarative or imperative languages), extensibility, adopted perspective (i.e., provider, consumer, or third-party), and maturity level (i.e., research prototype or commercial offering).

The literature review has been conducted in a thorough systematic manner. Combining key words (i.e., 'cloud', 'service', 'PaaS', 'cloud platform', 'monitoring', 'analysis', 'management', 'governance', 'adaptation', etc.), we searched for relevant papers on Google Scholar[1] – as a result, more than 200 papers have been identified. Next, paper abstracts have been analysed to decide how relevant surveyed papers are – approximately 50 papers remained at this stage. Finally, the remaining papers were thoroughly read and evaluated, which also resulted in identifying relevant authors and some additional papers, which were not included in the original collection of papers.

To provide a brief, yet informative overview of the relevant research efforts, we clustered the overall body of existing work into several groups and highlighted their distinguishing features. We believe, this way we provide the reader with sufficient knowledge of the current state of the art in the considered research area. Detailed descriptions for each of the studied approaches, along with a full explanation of the classification criteria, are summarised in Appendix D.

Accordingly, we identify five main clusters in the state-of-the-art research in cloud platform self-governance. We now summarise them, highlighting the most representative and relevant research works.

Cloud platform-native built-in monitoring solutions

These are already existing built-in solutions, developed and adopted by cloud platforms themselves. Representative examples of such approaches include Amazon CloudWatch[2], Rackspace Cloud Monitoring[3] (formerly known as CloudKick), and the recent Google App Engine Stackdriver Monitoring[4] .

[1] http://scholar.google.com/
[2] https://aws.amazon.com/cloudwatch/
[3] https://www.rackspace.com/cloud/monitoring/
[4] https://cloud.google.com/monitoring/

Amazon CloudWatch (Amazon Web Services, Inc., 2015) is a generic monitoring service, which is able to monitor the whole stack of AWS products, including Elastic Beanstalk, and display it as charts and diagrams. It primarily targets monitoring the infrastructure and application performance levels, which enables Elastic Beanstalk to automatically monitor users' applications and environment status. CloudWatch serves to monitor, manage, and publish various metrics, as well as configure alarm actions based on collected data from metrics. The collected information can assist customers in making operational and business decisions more quickly and with greater confidence. One of the benefits of Amazon CloudWatch is its support for user-defined extensions – these can be customised data metrics, alarms and notifications. For example, customised alarms help to implement decisions more easily by enabling users to send notifications or automatically make changes to the resources being monitored, based on rules they define. By extending the core functionality, customers are enabled to tailor the behaviour of the monitoring service to their personal requirements. Another advantage of CloudWatch is a management console with a run-time dashboard, through which users can get a real-time overview of all of their resources as well as alarms.

Rackspace Cloud Monitoring (Rackspace US, Inc., 2015) is a generic monitoring service for the whole stack of Rackspace cloud offerings, which is able to monitor standard infrastructure and application performance data metrics. The service supports two types of monitoring activities – namely, *remote* monitoring, which continually checks users' applications and platforms for external availability by testing connectivity from regional zones deployed throughout globally distributed data centers, and *agent* monitoring, which provides deeper insight into application resources, including servers and database instances. Additionally, similar to Amazon CloudWatch, Rackspace Monitoring allows users to define their own customised extensions, such as alarms and notifications, to meet individual user requirements.

Google App Engine provides the Stackdriver Monitoring service to enable its clients with visibility into the performance, uptime, and overall health of cloud-powered applications. Stackdriver (currently in Beta and offered as a free service) collects metrics, events, and metadata from Google Cloud Platform, hosted uptime probes, application instrumentation, and a variety of common application components including Cassandra, Nginx, Apache Web Server, Elasticsearch, and many others. Stackdriver ingests that data and generates insights via dashboards, charts, and alerts. Stackdriver also support user-customisable alerting and integration with various project management and collaboration platforms, such as Slack, PagerDuty, HipChat, and Campfire. One of the benefits of the Stackdriver

service is its support for hybrid and multi-cloud monitoring. At the moment, it also natively supports monitoring of AWS resources and services.

In most cases, approaches belonging to this cluster of related work are commercial and proprietary in their nature, with relatively little information available regarding design and implementation. Typically, such solutions provide an infrastructure- and application performance-oriented monitoring, alerting and visualisation – that is, they can measure virtual machine run-time execution in terms of CPU/memory utilisation, disk storage and network bandwidth. Arguably, proprietary solutions, due to their close and native integration with respective cloud platforms are expected to be effective, reliable, and easy to configure and use. Additionally, they are typically free to use, as their price is already included in the overall cloud service offering package.

Third-party monitoring frameworks

To a great extent, these solutions are similar to the previous cluster of cloud platform built-in approaches. For example, NewRelic,[1] Nagios,[2] and Datadog[3] provide a stack of monitoring solutions for a wide range of cloud platforms, including Heroku, MS Azure and AWS. On the other hand, there is Paraleap AzureWatch,[4] which is specifically designed to handle data monitoring in MS Azure. Apart from commercial offerings, we also identified a few open-source and academic solutions – namely, MonaLISA[5] and Zabbix[6]

[1] `http://newrelic.com/`
[2] `https://www.nagios.org/`
[3] `https://www.datadoghq.com/`
[4] `https://www.paraleap.com/AzureWatch/`
[5] `http://monalisa.caltech.edu/monalisa.htm/`
[6] `http://www.zabbix.com/`.

One difference of the surveyed third-party frameworks is that they are not necessarily designed to work with a single cloud platform. Such a cross-platform applicability enables cloud customers with a possibility of migrate their software from one cloud platform to another, while keeping the existing monitoring service. This advantage, however, results in a potential negative side effect – since third-party frameworks are not closely integrated with target cloud platforms (as opposed to built-in solutions), they can only focus on the application performance level, and do not always offer functionality to monitor the infrastructure level. In this sense, they remain 'third parties' both to cloud platform consumers and providers, who are not necessarily ready to share potentially sensitive information concerning the infrastructure resources with external services. As a result, third-party frameworks are typically oriented on data monitoring at the application performance level and provide customers with insights into web site utilisation statistics, response times, event logs, etc.

New Relic offers a whole stack of rich monitoring solutions, ranging from the level of data centre infrastructure resources to the level of individual applications and databases. A New Relic agent follows a non-intrusive approach to data collection and enables monitoring run-time application performance. 'Non-intrusiveness', however, does not allow it to observe data at the level of service composition, except for monitoring data storage service – New Relic provides a special agent for these purposes.

SOA solutions

These solutions, stemming from the SOC research, mainly target at the Service Composition and Coordination level of SBAs. They are not necessarily designed to address cloud platform-based SBAs, and even if they do, their primary focus is on service orchestration, rather than underlying resource consumption. Typically, approaches belonging to this cluster rely on inspecting a Business Process Execution Language (BPEL) definition of a Web service composition, which contains certain metrics and constraints for the monitoring component to check. Such constraints may include the order of service invocations, the format of exchanged messages, acceptable response rates, etc.

Nagios is a stack of multi-purpose monitoring frameworks, which enable resource and application performance monitoring based on an extensible architecture. It offers various monitoring services such as monitoring of host resources (e.g., CPU/memory utilisation, response times, etc.), network services and protocols (e.g., Simple Mail Transfer Protocol (SMTP), Transmission Control Protocol (TCP), HTTP, etc.). In the context of monitoring cloud-hosted SBAs, Nagios can only be used to monitor hardware resource consumption of individual applications and connected add-on services.

Paraleap AzureWatch is another third-party monitoring service, which supports monitoring MS Azure-hosted applications, and provides support for data visualisation, logging and alerting. With AzureWatch users can monitor resource consumption of Azure VMs and individual application statistics (e.g., CPU utilisation, requests/sec, bandwidth, and disk space). Moreover, it supports monitoring of MS Azure's built-in native services, such as SQL Azure (e.g., database size, open connections, or active queries) and Azure Service Bus, which is a messaging component in MS Azure. AzureWatch does not offer any kind of built-in support for third-party add-on services offered through the platform marketplace.

Heroku add-on services[1](Librato, New Relic Application Performance Monitor, Hosted Graphite, Pingdom, Still Alive, Blackfire.io, Dead Man' Snitch, Vigil Monitoring Service, Rollbar, Sentry, Bugsnag, RuntimeError, Honeybadger, Appsignal, Exceptiontrap, Airbrake Bug Tracker, Raygan.io, Informant, RedisMonitor) offer a wide range of monitoring tools form simple 'heart beat monitoring' to more sophisticated support for analysing fine-grained performance data. All these monitoring frameworks target at the application performance level, and do not offer support for monitoring metrics related to service compositions.

Datadog[2] is an application performance monitoring framework, which can be integrated with a range of cloud offerings, including Google App Engine and MS Azure. It can collect and aggregate a wide range of application performance metrics and display them to the user. Similarly to other third-party monitoring frameworks targeting at the application performance level, Datadog is not equipped with tools for monitoring data concerning third-party add-on services.

Monitoring Agents in a Large Integrated Services Architecture (MonaLISA) (Newman et al., 2003, Legrand et al., 2009) provides monitoring services to support control, optimisation, and management tasks in large-scale highly-distributed grid and cloud systems. To achieve this goal, it relies on a distributed network of agent-based services to support monitoring activities at several levels, including infrastructure and application performance. Even though its primary scope of application is grid computing, MonaLISA is also suitable for monitoring cloud environments at the infrastructure and application performance levels (but not at the level of service composition). An advantage of this approach is its extensible architecture, which relies on using distributed agents and multiple existing data collection mechanisms. As a result, MonaLISA is able to integrate new monitoring components in an easy and seamless manner.

An open-source framework *Zabbix* is an enterprise open-source monitoring solution, primarily focusing on networks and applications. It is build on a distributed architecture with a centralised web administration. The stack of Zabbix monitoring solutions includes agent and agentless tools for cloud service monitoring at the infrastructure and application performance levels.

Service Level Agreement Monitor (SALMon) (Ameller and Franch, 2008, Oriol et al., 2009, Leijon et al., 2008) – even though not directly targeting the cloud platform domain – aims at generic monitoring of heterogeneous web services, not just Web Service Description Language (WSDL)-based ones, detecting SLA violations and taking corresponding adaptive decisions. It fully implements the MAPE-K loop and relies on a novel domain-specific language to express the analysis and diagnosis policies. The SALmon domain-specific language serves to model the environment and overcome heterogeneity in service descriptions, which makes it relevant and applicable to our own problem domain. Moreover, the authors employ a streaming approach (i.e., a data stream management system) to data processing to achieve timely reactions. A potential limitation of the proposed approach, however, is that the authors focus only on the service composition level and do not take underlying infrastructure resources and their consumption into consideration.

The Web Service Level Agreement (WSLA) framework (Keller and Ludwig, 2003, Patel et al., 2009) serves to specify and monitor SLAs in SOA environments, including clouds. It employs an XML-based domain-specific language to define SLAs (i.e., resource- and business-related metrics) and constraints, which allows for extending the framework to cover all three layers of cloud computing. The resulting SLAs are then monitored and enforced by a dedicated enforcement component called SLA Compliance Monitor, which is publicly available as part of the IBM Web Services Toolkit. The declarative approach to defining SLAs enables smooth and seamless modifications, which also increases the potential of the framework to be extended.

IaaS-oriented approaches

These are solutions claiming to be oriented on the PaaS segment of cloud computing, but in reality typically rely on monitoring data metrics at the levels of service infrastructure and application performance.

Among the overall plethora of these approaches we want to focus on an approach by Nakamura et al. (2014), who propose utilising Semantic Web technologies (i.e., an OWL ontology and SWRL rules) to support self-management at the infrastructure level. The authors devised an ontology-based architectural model of a cloud data centre and delegated decision taking tasks to the SWRL reasoning engine. The proposed architecture also implements the MAPE-K reference model with a goal to maintain established SLAs and optimise resource consumption within an IaaS cloud. This idea of employing semantic technologies as an underlying analysis component is relevant to our own work. The difference is that

the authors are not considering the dynamic nature of the monitored data, and not employing a streaming approach to data processing (e.g., RDF stream processing). Also, the authors focus on the infrastructure level – an environment not as dynamic and rapidly changing as CAPs – and, therefore, do not discuss opportunities of the ontological model for being easily and transparently modified.

Another relevant work worth presenting is the framework called Monitoring Infrastructure using Streams on an Ultra-scalable, near-Real time Engine (MISURE) (Smit et al., 2013), which builds the monitoring infrastructure on the stream processing technology (i.e., S4 and Storm), thus making it even more relevant to our own work. The proposed framework can be seen as middleware, which integrates other existing infrastructure monitoring mechanisms (e.g., Ganglia, Simple Network Management Protocol (SNMP), log files, Amazon CloudWatch), which push monitored metricsfrom a wide range of heterogeneous cloud environments. The authors demonstrate performance of the framework, resulting from the usage of a streaming approach to data processing. The authors also argue that the proposed approach is extensible, as it follows a modular approach – i.e., custom Java extensions can be integrated into the framework to address emerging requirements.

Meng and Liu (2013), Meng et al. (2012, 2009) suggest a cloud monitoring framework offered as a service. In their paper, the authors consider an IaaS-oriented scenario, but claim that this Monitoring-as-a-Service solution potentially targets at all three levels of cloud computing. The approach primarily focuses on the infrastructure level, and main benefits include lower monitoring cost, higher scalability, and better multi-tenancy performance.

Katsaros et al. (2012, 2011) propose a cloud monitoring and self-adaptation mechanism, which spans across different segments of the IaaS level and collects monitoring data from application, virtual and physical infrastructure, and additionally considers the energy efficiency dimension. The authors also consider the PaaS segment as a an application scope for their research. Main benefits of the proposed approach include cross-level support for monitoring in the cloud paradigm, use of XML configuration files to define the self-adaptation logic, and extensibility of the proposed architecture.

JCatascopia Monitoring System (Trihinas et al., 2014) is an automated, multi-layer, interoperable framework for monitoring and managing elastically adaptive cloud services, developed by the University of Cyprus. It follows a non-intrusive approach to data collection, and primarily focuses on the IaaS segment of cloud computing. Additionally, it also offers the Probe API to implement customised extensions and metrics.

The Cross-Layer Multi-Cloud Application Monitoring-as-a-Service (CLAMS) framework (Alhamazani et al., 2015, 2014) represents and effort to create a monitoring solution spanning across cloud computing layers and several platforms. The proposed architecture can be deployed on several cloud platforms (e.g., the authors validated it on AWS and MS Azure) and is capable of performing QoS monitoring of application components. As far as the PaaS is concerned, the CLAMS's monitoring capabilities are limited to monitoring databases and application containers.

Admittedly, performing data monitoring and analysis at the infrastructure level is important, yet not enough for a truly PaaS-oriented monitoring solution. Relevant approaches belonging to this cluster tend to neglect the internal organisation and dependencies between add-on services within SBAs, treating them as 'black boxes'.

Truly PaaS-oriented approaches

This is a cluster of self-management solutions for cloud platforms, which we find most relevant to our own work. As opposed to IaaS-oriented approaches, these solutions focus specifically on the platform level and aim to provide a more holistic view to enable self-governance of software systems, deployed on (service-based) cloud platforms.

Breskovic et al. (2011) in their work focus on cloud markets – a subset of cloud platforms which are used for deploying software markets. Their goal is to create a cloud market platform, which "has the ability to change, adapt or even redesign its anatomy and/or the underpinning infrastructure during runtime in order to improve its performance". To do so, the authors also rely on the MAPE-K loop as a reference model and extend an existing data collection framework with cloud market-relevant sensors, which provide necessary data for performing analysis. The knowledge base includes a wide spectrum of rules and constraints, which are based on metrics belonging to the domain of e-commerce, such as, for example, price bids, number of active traders, cost of resource. Accordingly, based on these metrics, the autonomic manager then applies appropriate changes to the system to achieve predefined market goals. The authors, however, do not provide any evidence of whether their approach is generic and has the potential to be applied across a wide range of cloud platforms, not necessarily connected to the e-market domain.

Brandic (2009) focuses on the platfrom level of cloud computing and explains how cloud service SLAs can be described using novel domain-specific languages in terms of execution time, price, security, privacy, etc. to enable autonomic be-

haviour of a service. Based on these self-descriptive definitions, the proposed middleware platform is able to execute self-management within a cloud platform. The approach follows the MAPE-K loop, and claims to be generic and applicable to arbitrary cloud services – using an XML-based language facilitates framework extensibility.

Boniface et al. (2010) focus on the PaaS level of cloud computing, aiming to develop an automated QoS management architecture. In their paper, the authors focus on a cloud platform hosting a wide range of multimedia applications, which are monitored at various levels using the proposed software framework. The proposed approach utilises several modeling techniques (e.g., neural networks, Unified Modeling Language (UML), etc.) to detect potential SLA violations and support QoS. The authors argue that other techniques can be integrated if needed, thereby demonstrating the potential of the framework to be extended.

4.3 Identifying gaps in the state-of-the-art research

Having surveyed and clustered existing relevant work, we now attempt to share our generic observations from the studied materials. With this, we provide the reader with a critical view on the state of the art and identify existing gaps, which we will further address with our own approach.

Having briefly systematised the overall body of research efforts in the domain of monitoring and analysis of service-based cloud platforms, we now summarise and critically evaluate surveyed approaches with a goal to identify existing research and technological gaps in this domain. This will allow us to outline the application scope for our own research and the proposed EXCLAIM framework. While studying and classifying existing approaches, several observations were made. We now discuss each observation in more detail.

Focus on the IaaS-level metrics monitoring

First of all, there seems to be a research and technological gap in self-management mechanisms specifically targeting and operating at the PaaS level of cloud computing. Researchers tend to focus on the infrastructure level to enable scalability and elasticity, where admittedly have reached considerable results. The service-based level of PaaS, and particularly CAPs, is still under-explored.

There are approaches and solutions, which claim to be targeted at the platform level of cloud computing, but in fact depend on monitoring and interpreting low-level data, such as CPU/memory/disk utilisation and network bandwidth. In this

sense, they are not much different from the IaaS-level approaches dealing with resource elasticity and load balancing. Relying on universal and generic monitoring metrics, such solutions, therefore, are not designed to capture and interpret data related to the internal organisation and behaviour of deployed software. They treat user applications as 'black boxes' and can only measure its performance and behaviour in terms of (virtualised) hardware resources. Accordingly, their usage in the context of rich, service-based cloud platforms is limited, simply because potential problems may arise from the structure and interaction of SBAs, not the underlying hardware resources. For example, consider a worker process attached to a messaging queue and processing incoming jobs. A crash of this worker service will consequently lead to an overloaded messaging service. Obviously the problem exists at the level of the service composition and interaction and has to be solved by restarting or replacing the crashed worker service. Conversely, IaaS solutions will simply detect excessive resource consumption by the messaging queue, and will try to solve the problem by elastically provisioning additional space for queueing messages – a potentially inappropriate and even harmful solution in the given circumstances.

Existing approaches are isolated from each other

The third important observation is the existing isolation and heterogeneity of individual surveyed approaches. Each of them tends to solve a very specific and concrete problem related to self-management in clouds. For example, some of them focus on BPEL-defined Web service orchestrations; others try to predict workloads to support load balancing; some others try to detect network intrusions in physical and virtualised servers, etc. From this perspective, existing approaches are 'vertical', since they explicitly focus on a specific problem at a specific layer of cloud computing.

Even though the surveyed papers include sufficient explanation of how individual approaches are different from the rest, the papers hardly provide any insight of how they can potentially build on the existing similarities and overlaps; they do not suggest potential ways of how other relevant works can be re-used or combined so as to create a more diverse and effective solution. Quite frequently an existing IaaS-oriented solution can efficiently detect excessive hardware resource consumption, but cannot be extended so as to be able to detect problems at the level of add-on services and SBAs. In other words, existing solutions seem to exist in isolation from each other, which results in numerous approaches, which hardly overlap.

Since they all aim at (and succeed in) solving existing research gaps, this might

be acceptable from the scientific point of view. From a more practical perspective, however, these approaches hardly contribute to creating effective holistic self-governance capabilities within cloud platforms. Even though, when taken separately, surveyed solutions have the capacity to bridge existing gaps in cloud computing research and technology, there seem to be no mechanism, which would enable integration of these isolated approaches into a common multi-dimensional architecture.

In this light, we conclude that studied approaches are characterised with insufficient capabilities for extensibility. That is, possible modifications and enhancements lead to recompiling the whole system, which is caused by 'monolithic' (i.e., non-modular) implementations or usage of imperative programmng languages (i.e., hard-coding). Being good at solving one aspect of self-governance in cloud platforms, they are simply not designed to be extended so as to cover new emerging aspects.

Taken together, these considerations suggest that currently there is a need for a truly comprehensive architecture, which would enable synergies via integration of multiple approaches at various levels of cloud computing into a common framework. This leads to a requirement of creating an extensible framework, which would be capable of adjusting and configuringrespectively, so as to address emerging requirements if/when needed.

Analysis is typically not automated

Existing approaches targeting at run-time software performance tend to focus on collecting, synthesising and displaying data, rather than on performing sophisticated analysis over possible roots of a problem and planning corresponding adaptation actions. They are able to visualise and display various metrics (e.g., application response rate, number of invocations, network bandwidth, and memory usage) in a rich and easy-to-understand manner, but leave up to the customer to decide whether observed values may lead to a potentially critical situation. In other words, the analysis is done by the end user in a completely manual manner or in an interactive manner in situations, where customers are enabled to define their own simple policies for alerting – e.g., notifying the application owner whenever there is an increasing number of incoming network requests. On the other hand, there are several approaches, in which the analysis component is implemented in an automated manner. These approaches, tend to suffer from another shortcoming, as explained below.

Hard-wiring is present to a lesser or greater extent

Another observation from the conducted literature survey is the relatively wide adoption of interpreted (i.e., compiled) programming languages to implement self-management mechanism and analysis logic. To a lesser or greater extent, all of the approaches use some form of a knowledge base, where governance-related information is stored. Some approaches may rely on a rather hard-wired architecture, where information (e.g., policy constraints) is declared straight in the programming source code, and any modifications lead to software recompilations. On contrary, a BPEL definition of a Web service composition can be seen as an example of a knowledge base defined in a declarative manner. Despite the declarative definition of the knowledge base and analysis policies *per se*, enforcement of these policies still relies on hard coding of the required analysis logic to a lesser or greater extent. Arguably, in rather static and slowly evolving systems, such an approach might be sufficient. However, in more dynamic and rapidly changing service-based cloud platforms, frequent interruptions and code modifications might be unacceptable.

In the absence of such mechanisms at the platform level – that is, when such self-governance capabilities are not part of a PaaS offering – often, CAP providers expect software developers to implement this functionality themselves. This might result in rigid, hard-coded, task-specific and not-transferrable solutions. Clearly, there is a need for a truly generic mechanism which would apply to any software deployed on a CAP, irrespective of the business domain and underlying technology stack. Therefore, we claim that there seems to be a lack of truly loosely-coupled and declaratively defined analysis components, which would enable more flexible and rapid modifications of the self-governance mechanism in order to address emerging policies.

These presented observations outline a list of functional properties our own research approach has to demonstrate in order to fill the identified gaps and contribute to the state of the art in monitoring and analysis in service-based cloud platforms. In our view, the resulting solution has to operate on the PaaS level of cloud computing, but also take into consideration the IaaS and SaaS dimension – that is, to perform cross-layer monitoring and analysis activities within cloud platforms. As a prerequisite to this, the envisaged solution has to be extensible – that is, it has to be equipped with sufficient capabilities to adjust to emerging requirements if/when needed without major architectural modifications. For example, whenever a new add-on service is on-boarded to the cloud service marketplace, the self-governance mechanism has to be updated accordingly so as to integrate respective analysis and diagnosis policies in a seamless and transparent manner.

This requirement, in turn, leads to employing a declarative, loosely-coupled approach to declaring the knowledge base and policies. This concerns not just the definition of possible policies and constraints, but also employing a declarative approach to policy enforcement. In these circumstances, the analysis and planning algorithms are expected to be as easily modifiable as possible, and the degree of hard-wired coding has to be minimised. Extensibility also refers to the capacity of the envisaged solution to integrate other existing approaches so as not to 're-invent the wheel' given that an already existing, working and efficient solution for an emerging problem already exists.

Besides these considerations, the envisaged self-governance solution has to follow certain established practices for developing monitoring and analysis solutions. Such a mechanism is expected to perform monitoring and analysis activities in an automated and timely manner. Given the fact that applications and services hosted on a CAP are typically treated by the CAP providers as 'black boxes' with only APIs exposed to the outer world, the resulting solution has to be decoupled from the monitored entities and execute as a separate process, preferably in a dedicated execution space. The self-contained, proprietary nature of deployed applications and platform services also requires the monitoring mechanism to be designed and implemented in a non-intrusive manner. It has to rely either on the already-provided API entry-points or the surrounding context of the monitored entities, rather than instrument the monitored elements with additional intrusive data collection code. Since the resulting self-management mechanism is expected to be a part of the CAP, we assume that we adopt the CAP provider's perspective on the self-management activities with a goal to support stable operation of the cloud platform and to meet established SLAs.

4.4 Summary

In this chapter, our goal was to familiarise the reader with the state of the art in the research area of cloud self-governance in general, and data monitoring and analysis in particular, in the context of service-based cloud platforms. By doing so, we also outlined several research and technological gaps – that is, identified a scope for own proposed approach and listed initial high-level requirements for the future EXCLAIM framework. Briefly, the approach should focus on the service-based PaaS environments and employ a declarative language to define the knowledge base, including the architectural model of the managed environment, various policies and constraints, queries, etc. It also has to address the existing heterogeneity among add-on services and deployed application existing in CAPs.

The resulting solution has to be decoupled from the monitored entities and operate asynchronously with them to avoid possible bottlenecks. Another requirement is the near real-time operation, which will enable timely detection and reaction to potentially critical situations. The last, but not the least is the requirement to enable the framework with an extensible architecture which would enable it to address emerging requirements.

These considerations helped us to devise a novel approach to performing monitoring and analysis activities within CAPs. With these requirements at hand, we first designed a conceptual architecture of the EXCLAIM framework, and then implemented a prototype version to demonstrate viability of the proposed hypothesis. In the next chapters, we will discuss our proposed approach in more details.

Chapter 5

Conceptual architecture of the EXCLAIM framework

The goal of this chapter is to present the conceptual design of the EXCLAIM framework in a top-down manner. To do so, we first introduce a novel approach to treating individual components of CAPs, such as deployed applications and add-on services, as distributed logical software sensors, which can be potentially connected into a network so as to enable collected data to be monitored and analysed in one central location. Then, based on this fundamental interpretation, we outline a high-level conceptual architecture for the EXCLAIM framework. In doing so, we follow a top-down approach – we first describe the design of the future framework in a high-level and abstract manner, an then explain how it can be combined with SSW techniques so as to implement the prototype solution of the framework. The chapter also presents two enhancements to the core design of the EXCLAIM framework, which are aimed to improve its certain aspects. Section 5.3.1 explains how the core OWL ontology and the set of SWRL rules can be broken into several parts, thereby facilitating more loosely-coupled, modular structure, and what the benefits of doing so are. Section 5.3.2 introduces and explains the notion of criticality, which is a characteristic of platform add-on services, defining the particular set of detection rules to be applied in the given context. The higher the criticality, the more 'sensitive' rules to be applied.

5.1 Interpretation of Cloud Application Platforms as Sensor Networks

In Chapter 3.2 we explained the SSW research in general and the SSN ontology in particular in the context of physical sensor networks, such as, for example,

environmental monitoring systems. We also brought to the reader's attention the fact that the SSN ontology is a high-level and abstract vocabulary and is supposed to be extended with lower-level, domain-specific concepts. Accordingly, in this chapter, we will extend the traditional notion of sensors with the concept of logical software sensors, which can also act as sources of raw heterogeneous data to be monitored and analysed by a central component.

To introduce and explain this novel concept, let us first consider a hypothetical scenario, in which an SBA is deployed on a CAP and is leveraging add-on services provided by the platform. For example, Heroku, offers over 150 add-on services, and has over 1 million deployed applications (Harris, 2012). We also assume that deployed applications are typically coupled with one or more services. This claim is based on the assumption that customers deliberately choose Heroku as the target cloud platform for their software due to the extensive selection of easy-to-use add-on services, which facilitates rapid software development. That is, customers want to save their time and effort by re-using existing platform services, and that is why they deploy software on this cloud platform. Otherwise, they would not choose Heroku as the target cloud platform. Accordingly, in the considered hypothetical scenario, an e-commerce platform is running on Heroku and is supported with several add-on services for data storage, authentication, e-payment, search, notification, message queueing, etc. As this example suggests, the flexibility arising from the freedom to choose from a range of pre-existing services is appealing from the software developer's perspective – using just six services enabled to save considerable financial, time and human resources.

From the platform provider's point of view, however, offering this level of flexibility comes at a price. With add-on services replicated across multiple computational instances, and coupled with more than one million deployed applications, it becomes a challenging task to monitor the execution of the resulting CAP environment so as to detect failures and suboptimal behaviours. Maintaining the whole ecosystem at an operational level – that is, satisfying SLAs between the CAP provider and its consumers – is an inherently difficult challenge. A simplified diagram in Figure 5.1 schematically illustrates this situation when numerous deployed applications are deployed on the CAP and using various add-on services offered by the platform.

In the context of platform monitoring and maintenance, each of the elements depicted in the diagram is essentially an emitter of raw data, which can be further collected and analysed to enable CAP management tasks. For example, data storage services can provide data about available disk space, message queuing services can measure the throughput and utilisation of their channels and queues,

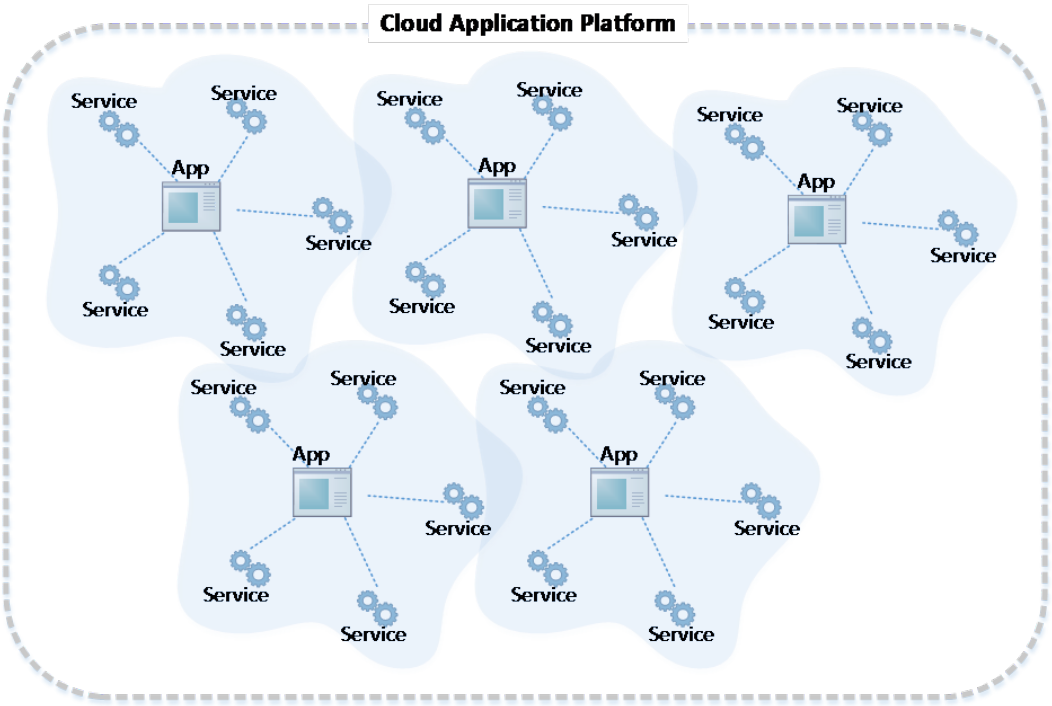

Figure 5.1: Applications are deployed on a CAP and coupled with the platform's add-on services.

and applications can provide statistics on the memory and CPU usage. The collected data, when properly structured, processed and analysed, can, for example, provide insights on how the resource consumption can be optimised. Accordingly, from this perspective, each of the elements of the CAP ecosystem can be seen as a software sensor node with several sensors attached to – in a manner, similar to physical sensor nodes with environmental sensors attached to them. Adopting this approach leads immediately to the idea that CAPs can be treated as sensor networks, and corresponding solutions and techniques can be applied.

By extending the notion of sensors to include not just physical devices, but anything that calculates or generates a data value – e.g., an application component, a database, or an add-on service – we can think of a particular service as a sensor and the whole platform as a network of such sensors. For example, we may be interested in monitoring response times from a platform add-on service which is connected to a user application (as illustrated in Figure 5.1). In this case, the service acts as a sensor node with several sensors attached to it, with the difference that unlike the traditional sensors, they are logical entities, represented with software functionality measuring and sending corresponding data values. Simply put, a sensor in our context is a piece of programming code, which measures

certain aspects of the sensor node – that is, of an add-on service. As we have seen in Chapter 3.2, sensor nodes may be equipped with one or more sensors. Accordingly, the add-on service in our scenario can be equipped with several logical sensors which measure, for example, its response time, number of incoming requests, current price, to name a few.

To enable the network with a global view and visibility into current states of its individual nodes, it is expected to have a central location, which would be responsible for data processing and storage. To do so, all these distributed software sensors, residing within a CAP, have to be connected into a network with a central processing location so as to enable them to transfer data for further analysis and interpretation. We try to illustrate this concept of a sensor-enabled CAP in Figure 5.2. The sensor nodes – add-on services, applications, platform components – are equipped with sensors, which are software components responsible for measuring certain characteristics, and connected into a network. Following the sensor network principles, values from a sub-network of monitored subjects may first go through a routing node – a software component responsible for transporting the values further to a central component and/or initial processing and aggregation of incoming information. Depending on the purposes of monitoring the central component may perform various functions, ranging from simple storage of monitored values to sophisticated analysis of those values, problem detection, or even execution of appropriate adaptation actions through a feedback mechanism.

5.1.1 Drawing parallels between sensor networks and cloud application platforms

While sensor networks are typically regarded as networks of distributed physical devices, which use sensors to monitor continually varying conditions at various locations (Baryannis et al., 2013), there are clear similarities with our own problem domain of data monitoring and analysis in service-based cloud environments. In our research, to support platform governance and maintenance, we are also facing the challenge of collecting heterogeneous data from multiple distributed sources and consequent querying this collected information to detect potentially critical situations. Accordingly, looking at CAPs from the Information Management point of view, the commonalities can be summarised as follows:

- **Volume**: as in sensor networks, the amount of raw data generated by deployed applications, components of the platform, users, services, etc. may be exceeding capabilities of the platform to manage them in a responsive and timely manner. Even if we do not take into consideration 'noise' – i.e.,

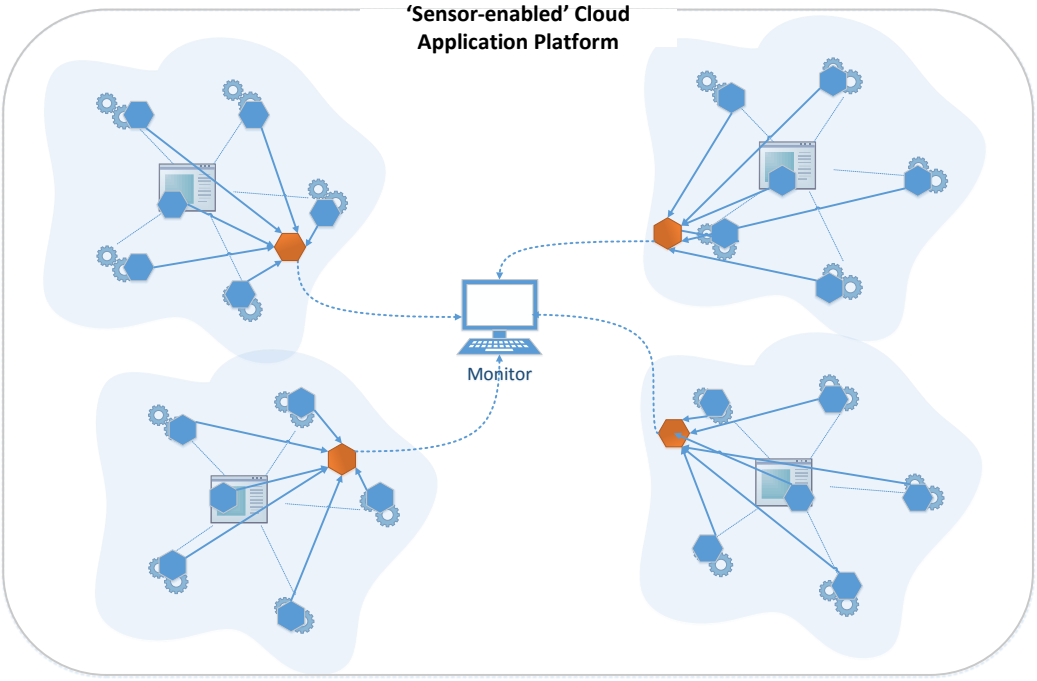

Figure 5.2: Schematic of a 'sensor-enabled' CAP.

information flows that are not immediately relevant for monitoring in the given context – the remaining amount of data is still considerable.

- **Dynamism**: in both sensor networks and service-based cloud platforms, various data sources are constantly generating raw values (which are then collected, processed, stored, deleted, etc.) at an unpredictable rate. Various platform components evolve, with new services being added and old ones removed, making the whole managed system even more dynamic.

- **Heterogeneity**: just as networked sensors can be attached to a wide range of different devices, so cloud platform data can originate from a wide range of distributed sources (applications, databases, user requests, services, etc.). This information is inherently heterogeneous, both in terms of data representation (different formats, encodings, etc.) and data semantics. For example, two completely separate applications from different domains with different business logic may store logging data in XML files. In this case, the data is homogeneous in its format – and potentially also in structure – but heterogeneous at the semantic level.

- **Distribution**: the information provided by both physical sensors and cloud platform components may come from various logically and physically dis-

tributed sources. On the logical side, platform data may originate from databases, file systems, running applications, add-on services, etc. and all these components may be physically deployed on distinct virtual machines, servers and even data centres.

Accordingly, these described similarities allow us to draw parallels between CAPs and problem domains, for which solutions proposed by the Sensor Web research community, based on sensor technology, have already been shown to be effective. Consequently, by drawing such parallels, we can build on work carried out by the Sensor Web community, and more specifically of the SSW community, in the context of dynamic monitoring and analysis of continuously flowing streams of heterogeneous data within CAPs. In the next sections of this chapter, we explain how the presented ideas were taken into account so as to outline the conceptual design of the future EXCLAIM framework.

5.2 Conceptual architecture of the EXCLAIM framework

In this section, we present the conceptual architecture of the EXCLAIM framework. The architecture relies on the two main 'pillars', presented to the reader in Section 2.2.1, and in the previous Section 5.1:

1. Interpretation of CAPs as networks of distributed software sensors
2. MAPE-K reference model for implementing closed adaptation loops

Accordingly, based on these two 'pillars', we will now describe our approach by sketching out a high-level conceptual architecture of the EXCLAIM framework (see Figure 5.3). In order to support both self-awareness and context-awareness of the managed CAP elements, we need to employ some kind of architectural model, which would describe the internal organisation and adaptation-relevant aspects of the cloud environment (e.g., platform components, services, available resources, connections between them, etc.). For these purposes we propose using an OWL ontology, which will contain the required self-reflective knowledge of the system. Inspired by the SSN ontology, such an architectural model, represented with OWL, will serve as a common vocabulary of terms, shared across the whole EXCLAIM framework. Conceptually, we distinguish 3 main elements of the framework: triplification engine, continuous SPARQL query engine and OWL/SWRL reasoning engine. Accordingly, our ontological classes and properties will serve as building blocks for creating RDF streams, Continuous SPARQL (C-SPARQL) queries and SWRL rules, used consequentially in each of these components respectively.

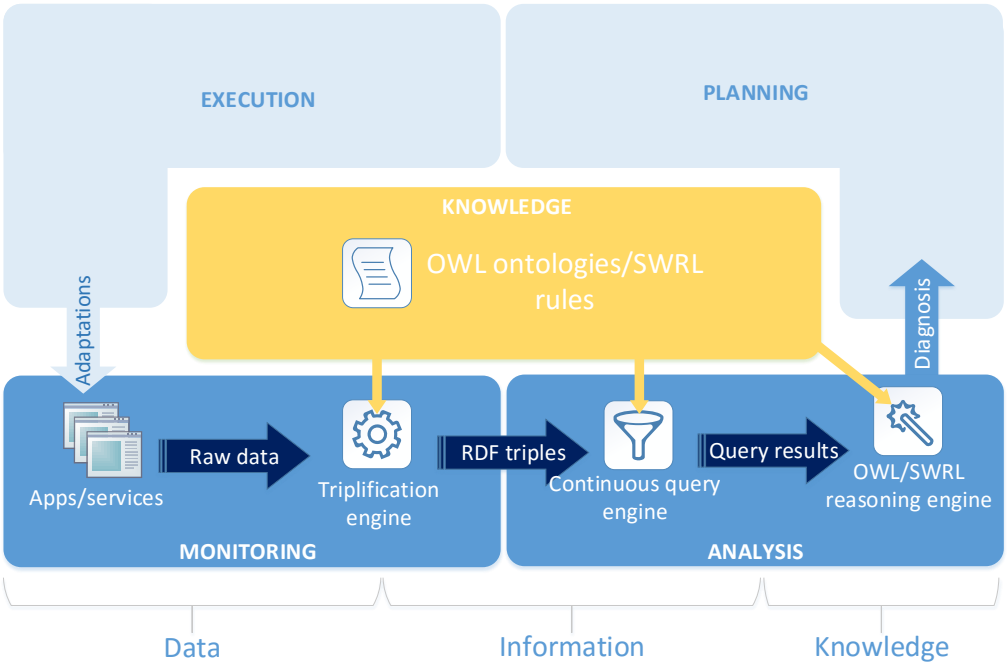

Figure 5.3: Conceptual architecture of the EXCLAIM framework (based on the MAPE-K reference model).

1. The **triplification engine** is responsible for consuming and 'homogenising' the data generated by deployed applications, platform components, add-on services, etc. That is, the main function of this is component is to transform raw data into information by representing collected heterogeneous values (e.g., JSON messages, SQL query results, and text files) using a single uniform format. Accordingly, the engine takes as input streams of unstructured heterogeneous data, transfroms them into semi-strucutred RDF triples using the ontological vocabulary, and generates streams of RDF triples. Using RDF as a common format for representing streaming data, and OWL concepts as subjects, predicates and objects of the RDF triples, allows us to benefit from human-readability and extensive support of query languages.

2. The next step of the information flow within our framework is the **continuous SPARQL query engine**, which takes as input the flowing RDF data streams generated by the triplification engine and evaluates pre-registered continuous SPARQL queries against them, to support situation assessment. In the first instance, situation assessment includes distinguishing between

usual operational behaviour and critical situations by matching them against critical condition patterns. The process of encoding patterns is iterative – that is, new critical conditions have to be constantly added to ensure the pattern base remains up to date. For example, the EXCLAIM administrator (i.e., the CAP provider) and add-on service providers may need to add corresponding queries for newly-added add-on services, or remove existing ones. Depending on circumstances, existing queries and critical thresholds also may need to be adjusted with respect to the current situation and system workload.[1] We propose using one of the existing continuous SPARQL languages to encode critical condition patterns. By registering an appropriate SPARQL query against a data stream, we will be able to detect a critical situation with a minimum delay – the continuous SPARQL engine will trigger as soon as RDF triples on the stream match the WHERE clause of the query[2]. To some extent, this step can be seen as a filtering stage, and the outcome of this component is filtered information – a sub-set of RDF triples, which represent potentially critical situations. The querying engine will only trigger in response to potentially critical situations, and thereby will save the static reasoning engine from processing numerous non-critical, 'noisy' tuples. 'On-the-fly' processing of constantly flowing data, generated by hundreds of sources, as well as employing one of the existing RDF streaming engines, are expected to help us in achieving near real-time behaviour of the EXCLAIM framework.

3. Once a potentially critical condition has been detected, a corresponding confirmation has to be generated (and appropriate adaptation actions can be planned and executed) by the **OWL/SWRL reasoning engine**. This step requires not just checking several simplistic 'if-then' statements, but rather requires more complex and sophisticated reasoning over the possible reasons for a problem, and identification of potential adaptation strategies. In this respect, we envisage addressing this challenge (at least partially) with SWRL rules, which provide a sufficient level of expressivity for defining policies, and are expected to exempt us from the effort-intensive and potentially error-prone task of implementing our own analysis engine from

[1]Machine learning techniques may be employed in this context to assist with maintaining the list of critical condition patterns. More details on this can be found in Section 9.2.

[2]With SPARQL as an underlying query language, in the future it is also possible to benefit from inference capabilities – that is, apart from just querying data, we may also be able to perform some reasoning over RDF triples. The reasoning capabilities, yet immature, depend on the entailment regime of the continuous SPARQL language to be selected, and respective tool support. For more details, please refer to Section 8.3.

scratch. Accordingly, we will rely on the built-in reasoning capabilities of OWL ontologies and SWRL rules, so that the routine of reasoning over a set of possible alternatives is done by an existing, tested, and optimised mechanism. The outcome of this reasoning step is a confirmed occurrence of a detected critical situation.

When defining detection policies with OWL and SWRL, we, as application developers and platform administrators, may benefit from the following (Dautov et al., 2012) (we will further discuss potential benefits of our proposed approach in Chapter 8):

- **Separation of concerns** – with declaratively-defined ontologies and rules separated from the actual platform/application programming code, it is easier to make changes to detection policies 'on the fly' (i.e., without recompiling, redeploying and restarting the platform/application) in a transparent seamless manner.

- **Extensibility** – separation of concerns, in turn, opens opportunities for creating an extensible and easily modifiable architecture for defining the knowledge base. To reflect emerging requirements in a timely manner, new concepts can be easily added to the OWL ontology, and then respective SWRL policies can be defined and added to the policy base. Same applies to the reverse process – if needed, outdated definitions can be removed at any time with minimum effort.

- **Increase in reuse, automation and reliability** – once implemented and published on the Web, an ontology is ready to be re-used by third parties, thus saving ontology engineers from 'reinventing the wheel'. Moreover, since the reasoning process is automated and performed by a reasoning engine, it is expected to be free of so-called 'human factors' and therefore more reliable.

5.3 Enhancements to the main conceptual design

Having introduced the main conceptual design of the framework, we now explain several additions to it. These additions are intended to improve certain aspects of the EXCLAIM framework and aim at increasing its support for i) modular and declarative definition of the knowledge base; and ii) flexible and fine-granular monitoring of individual services and applications.

5.3.1 Modularity and self-containment

One of the main goals of the presented research work is to enable a declarative and loosely-coupled approach to defining and modifying detection policies. We aimed at separating definition of policies from their actual enforcement. For this reason, we followed the SSW approach, which enabled us to define the knowledge base in a declarative manner by means of OWL ontologies and SWRL rules. However, as the resulting knowledge base is expected to grow in size and complexity, it may result in a considerably heavy-weight ontology, and therefore will slow down the reasoning process, which will try to infer the required information with respect to all facts found in the OWL ontology and SWRL rules. Simply put, the more 'heavy-weight' and richer the knowledge base gets, the slower the reasoning process becomes.

In these circumstances, application of the modularity principles might be of potential benefit – that is, breaking down the whole knowledge base into multiple parts, so that only necessary and relevant facts are taken into account when performing reasoning activities, while the unnecessary bits are simply left aside in the given reasoning context. Modularity, which is known to be one of the most widely-spread approaches to address the ever-increasing complexity in computer systems[1] (Baldwin and Clark, 2000, 2003), has the potential to become a key to success. Using ontology engineering terminology, this means that the terminology box (i.e., the TBox) of the ontology has to be minimised (Gruber, 1993). As opposed to the axiomatic box (i.e., the ABox), which contains actual instances of the TBox classes, the TBox declares classes and properties. Decidability and computability characteristics of an ontology mainly depend on the TBox, and minimising the amount of statements in the TBox is typically expected to speed up the reasoning process.

Another reason for employing the modular architecture is the necessity to enable third-party service providers with capabilities to modify detection policies. So far, we assumed that CAP providers are responsible for maintaining governance-related knowledge, which concerns not only the internal platform components, but also third-party services, which are registered with the given CAP service marketplace. In reality, however, this is not necessarily the case. Typically, third-party service providers, having deployed their software on a cloud and exposing the API to the users, take on the responsibility to maintain the software and associated resources, and provide customers with required support. This means that

[1]For example, SOA can also be seen as an example of the modular approach, which helps to tackle the ever-growing complexity of modern enterprise systems by minimising the number of system components and splitting them among interested parties.

CAP providers treat third-party services as 'black boxes' and need not be aware of their internal architecture and organisation. Accordingly, this may result in a situation, where governance policies are incomplete, imprecise, or even invalid, which in turn may lead to incorrect diagnoses, non-optimised resource consumption, system failures, and, eventually, put platform stability at risk.

As a potential solution to this problem we are employing Linked Data principles. The primary goal of Linked Data is to enable discovery and sharing of semantically-enriched data over the Web using standardised technologies, such as URIs and RDF (Bizer et al., 2009). By adding references and interlinking individual datasets published on the Web, it unites them into a giant global graph. In other words, Linked Data implies the ubiquitous re-use of existing distributed data, which is exactly what we need in order to separate various pieces of adaptation policies between CAP owners and third-party service providers.

Utilising Linked Data principles has the potential to create an extensible architecture where a cloud sensor network consists of independent self-contained sensors (i.e., platform components and services), described by a two-tier distributed set of interacting ontologies. To implement the described modular approach to the definition of the knowledge base, we employ the following two-tier architecture:

1. **Core OWL ontology**, which contains all the necessary concepts, relations and default SWRL rules needed to define the default governance-related behaviours of platform components and services. It is expected to be a rather static entity and stay under control of the platform administrators.

2. **Linked SWRL extensions** are a set of linked sets of SWRL rules developed by third-party service providers and published on the Web. They specify detection and adaptation policies for respective services registered with the CAP marketplace. These extensions may either extend or overwrite the default behaviour specified in the core ontology. These extensions only apply to individual services – that is, they are only added to the knowledge base, whenever a respective service is involved in the given monitoring and analysis scenario.

5.3.2 Criticality levels, criticality dependencies and application profiling

So far, we have thought of and described add-on services and user applications coupled with them, in a rather abstract manner, without considering certain real-world aspects and assumptions. We have been treating individual add-on services and resulting SBAs as entities, equally deserving to be monitored and managed.

In practice, however, it can often be the case that some services (and therefore – depending applications) need more attention, whereas the others need it to a lesser extent. In other words, we can distinguish and classify deployed applications with respect to their criticalities. A criticality is a context-dependant qualitative characteristic of add-on services, which determines which detection policies – strict or loose – have to be applied in a particular case.

For example, imagine a service-based composition deployed on a CAP, which actively uses messaging queues as a means of communication between application subcomponents. Being heavily dependent on the messaging queue service, the considered application cannot tolerate a situation where the service crashes or underperforms – in such circumstances, it will simply stop functioning. Such an application, therefore, can be classified as critical with respect to its messaging queue service, and therefore, when identifying critical situations, critical thresholds have to be defined as low as possible. On the other hand, the very same application may be using an Short Message Service (SMS) notification service for sending notifications to its users. This service, in the context of the given application, is not that critical, because, for example, other ways of user notification exist. It means that the application can tolerate a temporary outage of the SMS notification service and still be functional. Accordingly, the SMS notification service in the given scenario is not critical, and thereby loosest detection policies can be applied.

It should be noted that the notion of service criticalities is context-dependant and only exists in association with criticalities of resulting service-based compositions. One and the same service may be critical in the context of one service-based application and non-critical in the context of another. Coming back to the previous example, the SMS notification functionality may lie at the core of a dedicated online SMS sending service,[1] and the outage of the corresponding service, even for a few minutes, may be lethal to such kind of SBAs.

Accordingly, to describe a service-based application in terms of criticalities of the constituting services, we are introducing the concept of application profiling. Essentially, an application profile is a key-value mapping, where key is a depending service, and value is a corresponding level of criticality for that service.

It is also possible that not only user applications are dependent on certain services, but services themselves can be interdependent on each other (see Figure 5.4). Such a situation is quite common – for example, as explained in Chapter 6, a messaging service is used to send payload which is then to be persisted to a database, provided by a data storage service. In the described scenario, messaging queue and database services are dependent on each other. Accordingly, it is also

[1] See, for example, `http://www.afreesms.com/`.

possible that these services might have been given different criticalities by the user, which may result in potentially ambiguous and even dangerous situations. To prevent them, we introduce the notion of criticality dependencies. Whenever a dependency with such a criticality level mismatch is detected, we apply the highest criticality level found among the dependent services.

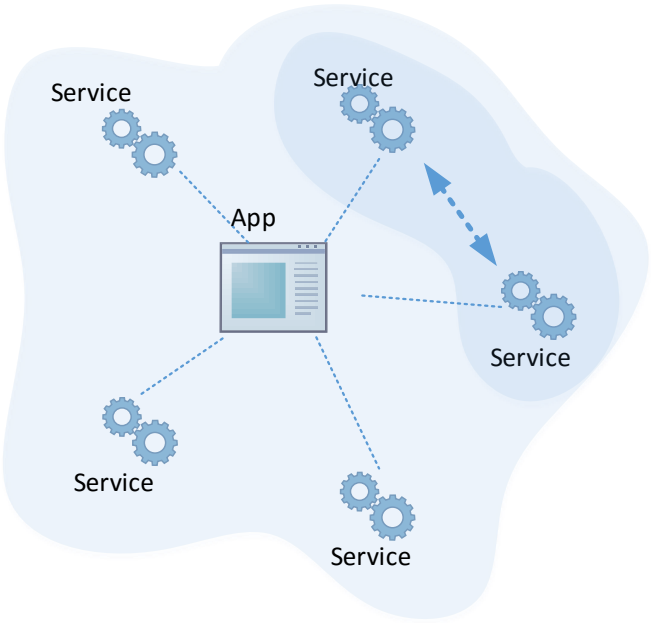

Figure 5.4: An application is dependent on several add-on services, two of which are also interdependent between each other.

In the context of the presented research work, the concepts of criticality and application profiling are used to enable a more fine-grained control of the monitoring and analysis activities. By assigning low criticalities to services and SBAs and applying respective policies, we are able to decide whether an observed situation represents a threat and therefore requires additional system resources to be used so as to perform further planning and adaptation activities. In other words, we enable the EXCLAIM framework with 'filtering' capabilities, which can serve to save computational resources – that is, by minimising the number of false positives (i.e., normal situations classified as critical due to low critical thresholds), we avoid unnecessary further actions.

To conclude, it is needs to be noted that the concept of application profiling can be seen as a communication channel between CAP consumers and CAP providers. Consumers are not allowed to interfere with the back end of the EXCLAIM frame-

work, including the knowledge base, but are essentially the ones who know the best how critical their applications are. That is, they are aware of the internal structure and business logic of their applications, and therefore might want to make the platform aware of what aspects of their software systems need to be cared after more thoroughly. Accordingly, by submitting an application profile, they let the platform know of how critical the software is. The platform, in its turn, will then be able to treat deployed software and connected add-on services with more care if needed, based on the submitted criticality profile.

5.4 Summary

The main goal of this chapter was to familiarise the reader with the conceptual architecture of the EXCLAIM framework. This architecture is underpinned by the two main concepts. The first influencing concept is the novel interpretation of CAPs with their add-on services and deployed applications as distributed networks of logical software sensors. This analogy is based on the similarities between the problem domains, where solutions from the Sensor Web research area proved to be useful, and the problem domain of CAP data monitoring and analysis. The second concept is the MAPE-K reference model for implementing self-adaptation loops and creating autonomic systems. The design of the framework primarily focuses on the Monitoring and Analysis components of the MAPE functionality with a possibility to be extended or integrated with existing solutions so as to achieve the complete MAPE-K functionality. Having introduced the design of the framework and additional enhancements, the next chapter proceeds with implementation details of the EXCLAIM framework.

Chapter 6

Implementation details of the EXCLAIM framework

Having introduced the conceptual architecture of the EXCLAIM framework in the previous chapter, we now continue with the top-down approach and discuss the implementation details. We describe each of the main three components of the framework – namely, the triplification engine, the continuous query engine and the reasoning engine. In all three of these components, a key role belongs to the ontological model, which provides a shared common vocabulary of terms to define RDF triple streams, continuous SPARQL queries, and SWRL rules respectively. Therefore, a dedicated section will explain the structure and the role of the CSO. The concepts explained in this chapter, will be further extended in the next chapter, which puts together the design and implementation details of the framework and demonstrates how the framework functions based on a use case scenario.

6.1 Overview of technical details

The prototype version of the EXCLAIM framework is a Java project, which was developed using Eclipse IDE with an installed Heroku plugin. This plugin facilitates the process of deploying and upgrading an application to Heroku directly from within Eclipse, as opposed to the traditional command line-based way of packaging and uploading applications on the CAP. All Heroku apps are required to be enabled with the Maven[1] nature to support the application building process and resolve library dependencies.

[1] Apache Maven (`https://maven.apache.org/`) is a build automation tool primarily for Java applications. It automates the process of software compiling and packaging based on a descriptions of how software is supposed to be built, and its dependencies.

As of April 2016, there are 11 packages, 43 classes, and 3,646 lines of code in the EXCLAIM framework. Apart from the standard libraries, constituting the Java Software Development Kit (SDK), several external libraries were used to implement the project. Table 6.1 summaries the most important of them.

Table 6.1: Third-party JARs and their role in the implementation of the EXCLAIM framework.

Libraries and JARs	Description	Usage
OWL-API[1]	The OWL API is a Java interface and implementation for the OWL. The latest version of the API is focused on OWL 2 which encompasses OWL-Lite, OWL-DL and some elements of OWL-Full. Main features of OWL API include: • API for OWL 2 and an efficient in-memory reference implementation • RDF/XML parser and writer • OWL/XML parser and writer • OWL Functional Syntax parser and writer • Turtle parser and writer • Support for integration with reasoners such as Pellet and FaCT++ • Support for black-box debugging	OWL API For manipulating the CSO and SWRL rules. It provided methods for creating, modifying, and deleting entities in the CSO and linked SWRL extensions. Besides that, thanks to its integration with the Pellet reasoner, it supported all reasoning-related activities within the framework. OWL API also took care of the importing mechanism, which was necessary to support the modular architecture based on the core CSO and multiple SWRL extensions.
Pellet[2]	Pellet is an open-source Java-based OWL 2 reasoner. It can be used in conjunction with the OWL API library. It incorporates optimisations for nominals, conjunctive query answering, and incremental reasoning.	The Pellet reasoner was used to enable reasoning by the OWL-API. While developing the CSO with Protege, we used Pellet as the underlying reasoner to test the ontology. Therefore, the same reasoner (even the same version) was used in the framework, to make sure the reasoning results are consistent with Protege.

[1] `http://owlapi.sourceforge.net/`
[2] `https://github.com/complexible/pellet/`

C-SPARQL Ready-to-Go Pack[1]	The ready-to-go pack is a Java library, developed by the Polytechnic University of Milan on top of the Erlang streaming engine, which contains an implementation of the C-SPARQL RDF processing engine. C-SPARQL queries consider windows, i.e., the most recent triples of such streams, observed while data is continuously flowing.	We use the C-SPARQL engine library to enable the autonomic manager with capabilities to process RDF streams. C-SPARQL querying is designed to be a filtering step before the actual reasoning, which is quite computationally demanding. In other words, we cannot afford launching the reasoning process for every single triple observed on the stream, but have to minimise their number, thus running the reasoning process less frequently.
Apache Jena[2]	Apache Jena is an open source Semantic Web framework for Java. It provides an API to extract data from and write to RDF graphs. The graphs are represented as an abstract 'model'. A model can be sourced with data from files, databases, Universal Resource Locators (URLs) or a combination of these. A model can also be queried with SPARQL. Jena also provides support for OWL. The framework has various internal reasoners and the Pellet reasoner can be set up to work with Jena. Jena supports serialisation of RDF graphs to: • a relational database • RDF/XML • Turtle • Notation3	Apache Jena was used to handle all RDF-related activities. This library was also a fundamental dependency to enable proper functioning of the OWL API library and the C-SPARQL Ready-to-Go pack.

[1] `https://github.com/streamreasoning/C\gls{sparql}-ReadyToGoPack/`
[2] `http://jena.apache.org/`

Gson[1]	Gson is a Java library that can be used to convert Java objects into their JSON representation. It can also be used to convert a JSON string to an equivalent Java object. Gson can work with arbitrary Java objects including compiled objects, which do not necessarily have their source code available. Its main features are: • Provide simple toJson() and fromJson() methods to convert Java objects to JSON and vice-versa • Allow pre-existing unmodifiable objects to be converted to/from JSON • Extensive support of Java generics • Allow custom representations for objects • Support arbitrarily complex objects (with deep inheritance hierarchies and extensive use of generic types)	Among other existing alternatives for JSON serialisation, the Gson library is known to be most suitable for small-size objects (Alin, 2015). This feature makes it a good choice for handling relatively small RDF triples and support the process of exchanging data within the EXCLAIM framework. Primarily, Gson was used to serialise triples when exchanging data over RabbitMQ queues.
RabbitMQ[2]	RabbitMQ is open source message broker software that implements the Advanced Messaging Queue Protocol (AMQP). The RabbitMQ server is written in the Erlang programming language. Client libraries to interface with the broker are available for all major programming languages, including Java.	RabbitMQ was used as the main means of collecting monitored values from software sensors. Therefore, the client library was used for both sending RDF data from the software sensors and receiving it by the EXCLAIM framework.

The described Java technologies helped us build the prototype version of the framework. Initially, it was a command-line tool, but at the later stage a graphical user interface was added to make the application more user-friendly and easy-to-use. The graphical interface is a simple management console accessed through the Web browser (see Figure 6.1).

As it is seen from the screenshot, the functionality of the EXCLAIM frame-

[1] https://github.com/google/gson/
[2] https://www.rabbitmq.com/

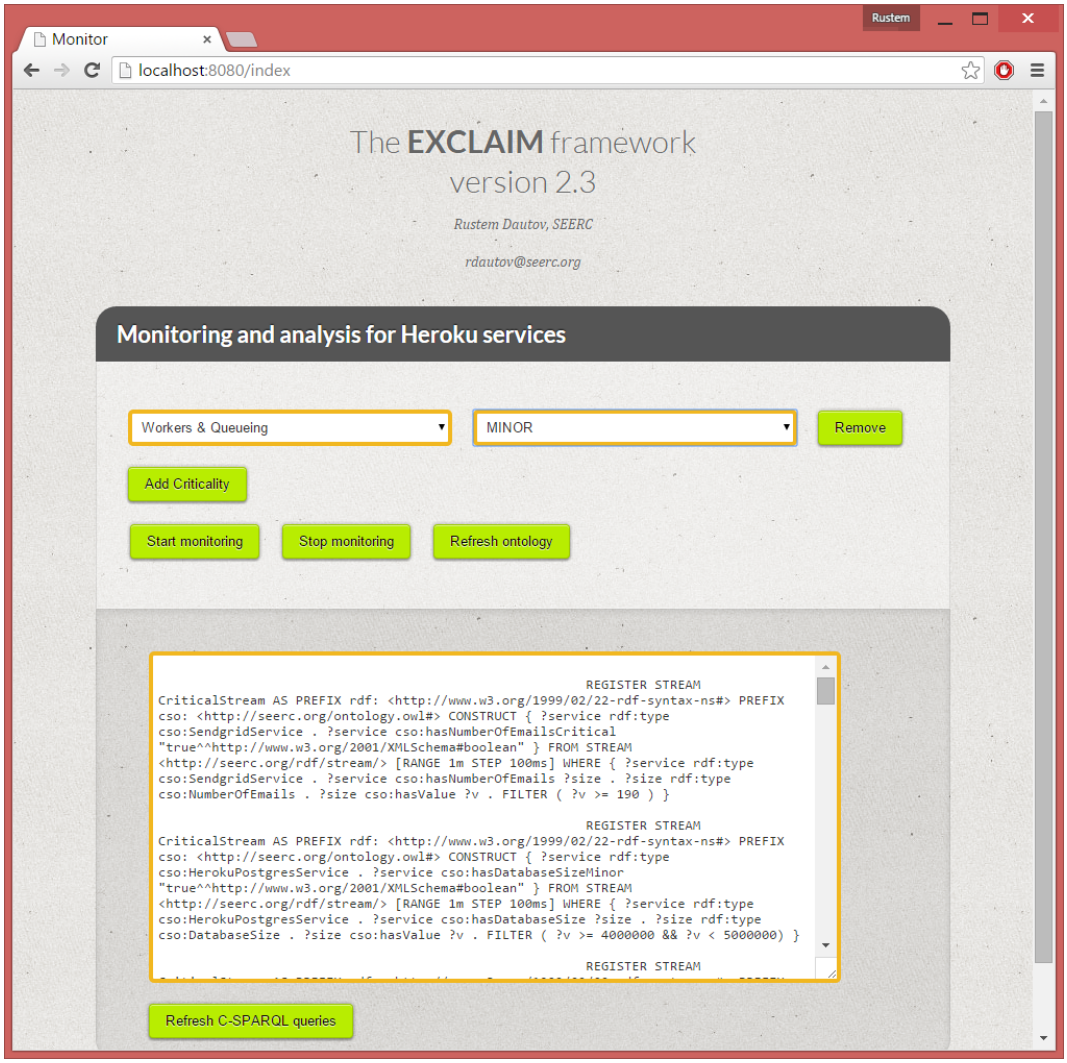

Figure 6.1: Management console of the EXCLAIM framework.

work can be accessed via main 5 buttons. Table 6.2 summarises these functional elements and describes their role within the framework.

Table 6.2: Main features of the EXCLAIM framework accessed via the management console.

Feature	Description
Add criticality	With this component, the user is enabled to profile applications to be monitored. By specifying criticality levels for corresponding add-on services the monitored application is connected to, the user creates an application profile, which is then parsed by the framework and respective diagnosis and adaptation policies are applied.

Start monitoring	With this button, the user can start the monitoring and analysis process.
Start monitoring	With this button, the user can stop the monitoring and analysis process.
Refresh ontology	With this button, the user can refresh the knowledge base of the EXCLAIM framework. One of the main goal of the research work presented in this document is to create a mechanism for declarative definition of policies. Respectively, it is possible to modify the core CSO and linked SWRL extensions at run-time, and by clicking this button to make the framework reason over this newly-added knowledge base.
Refresh C-SPARQL queries	With this button, the user can refresh C-SPARQL queries. Similar to refreshing the ontology, this function was inspired by the declarative approach to defining knowledge within the framework. Accordingly, it is possible to modify the C-SPARQL queries at run-time and make the framework transparently switch to using the new ones. The text field in the management console, once the monitoring and analysis process is started, returns the list of all C-SPARQL queries currently constituting the knowledge base.

6.2 Implementation details

We now proceed with a detailed explanation of how the EXCLAIM framework prototype was implemented. This explanation is aligned with the main components of the conceptual architecture presented in the previous chapter and accompanied by several code snippets, which are intended to demonstrate how the key functions are implemented within the framework.

6.2.1 Triplification enegine

The triplification engine is the initial step of the data flow within the EXCLAIM framework, and is the main element through which managed applications and services interact with the framework. Its main responsibility is to 'extract' raw data from the managed elements and transform it into semantic RDF triples using the CSO vocabulary.

Following the non-intrusive principle to data collection, this component relies on already existing APIs provisioned by services to gather required metrics. For example, in order to obtain the most recent view on the current utilisation of an RDBMS service (e.g., occupied space, number of client connections, current backup processes, etc.) it is necessary to run an SQL query on an auxiliary statis-

tics table[1] for system administration purposes, which contains run-time meta-data about the database. The Postgres default rate for updating the statistical information is 500 ms, and can also be configured to provide a more up-to-date view on the current state of the database. Accordingly, the sampling rate of the EXCLAIM framework for data extraction should be configured respectively so as to operate in an optimised manner – that is, fast enough to provide actual data and at the same time avoid redundant (and expensive) calls to the database.

The following SQL snippet, for example, fetches the current values for the occupied space (i.e., `SIZE`) and the number of established client connections (i.e., `COUNT`) to the database `destinator_db` belonging to the user `destinator_user`.

Listing 6.1: SQL query snippet.

```
SELECT pg_database_size('destinator_db') AS SIZE, COUNT(*) AS COUNT FROM
    pg_stat_activity a WHERE a.usename='destinator_user'
```

Other services, such as IronWorker service, expose its run-time meta-data via a RESTful management API, accessible by the client libraries. For example, to get the current number of jobs in the IronWorker queue waiting to be processed as an array of JSON objects, it is necessary to invoke a corresponding API function (see Listings 6.2 and 6.3).

Listing 6.2: IronWorker API function to get the number of tasks in the queue.

```
public List<TaskEntity> getTasks(Map<String, Object> options) throws \gls
    {api}Exception {
JsonObject tasks = api.tasksList(options);
List<TaskEntity> tasksList = new ArrayList<TaskEntity>();
for (JsonElement task : tasks.get("tasks").getAsJsonArray()) {
tasksList.add(gson.fromJson(task, TaskEntity.class));
}
return tasksList;
}
```

Listing 6.3: Sample client code to fetch the number of tasks to be executed by the IronWorker service.

```
options = new HashMap<String, Object>();
options.put("running", 1);

tasks = IronWorkerManager.getClient().getTasks(options);
```

Similar standard mechanisms for extracting data metrics exist for all the other add-on services, provided by the cloud platform marketplaces (including Heroku).

[1] In Postgres Database Management System (DBMS), this table is called `pg_stat_activity`. Similar tables exist in the majority of other relational databases.

An important benefit of the data extraction used by the EXCLAIM framework is that there is no intrusion to the source code of the monitored services and applications. The framework only requires user credentials to get access to individual instances of services – an acceptable requirement given that the EXCLAIM framework is assumed to be part of the cloud platform and act as a trusted entity for the consumers.

Once the raw data is extracted, it then needs to be uniformly represented using the CSO 'building blocks'. At the moment, this is mainly a manual process, which involves mapping between the source raw data and target semantically-annotated triples.[1]

It needs to be explained that a single raw value is represented with multiple RDF triples, which form an RDF graph. Depending on the requirements, additional RDF triples serve to provide a more unambiguous and context-aware information to the EXCLAIM framework. For example, Listing 6.4 below demonstrates how the number of current client connections to the PostgreSQL service (i.e., the service `postgres-service-06` has 12 client connections) is translated into the RDF representation to be further processed by the EXCLAIM framework.

Listing 6.4: A single raw value is represented using four RDF triples.

```
cso:postgres-service-06 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasNumberOfConnections cso:number-of-
    connections-122
cso:number-of-connections-122 rdf:type cso:NumberOfConnections
cso:number-of-connections-122 cso:hasValue "12"^^xsd:int
```

Accordingly, the newly-generated RDF triples are serialised into string objects and sent to the messaging queue, from where they are picked up by the EXCLAIM framework, de-serialised into RDF triples and processed by the C-SPARQL query engine. In the current version of the EXCLAIM framework, Heroku's RabbitMQ messaging service is being used to facilitate information transfer between data connectors and the core EXCLAIM framework. Using RabbitMQ's client API the process of configuring a message queue (see Listing 6.5) and sending a message (see Listing 6.6) can be accomplished in several lines of code.

Listing 6.5: Declaring and initialising a RabbitMQ service queue.

```
String uri = "amqp://guest:guest@localhost";
ConnectionFactory factory = new ConnectionFactory(uri);
```

[1]There are already existing tools for converting data stored in relational databases into the RDF representation, using special mapping languages (e.g., RDB to RDF Mapping Language (R2RML) – `http://www.w3.org/TR/r2rml/`), which may simplify the triplification process as far as relational data storage services are concerned. Similarly, analogous techniques can be envisaged for RDF triplification of JSON, XML and other formats.

```
Connection connection = factory.newConnection();
Channel rabbitMqTaskChannel = connection.createChannel();
rabbitMqTaskChannel.queueDeclare("exclaim_queue");
```

Listing 6.6: Sending a string message to the RabbitMQ queue.

```
String message = "cso:number-of-connections-122␣cso:hasValue␣'12'^^
    xsd:int"
rabbitMqTaskChannel.basicPublish("exclaim_queue", null, message.getBytes
    ());
```

Please note that individual 'connectors' responsible for raw data extraction and RDF transformation can be configured to function at various sampling rates. Finding the right sampling rate may depend on various criteria, such as the update rate of the source raw data (e.g., table `pg_stat_activity`), amount of available computational resources to execute frequent data extraction and transformation, tolerance of the managed services and applications to potential delays, etc.

Also, as it is seen from the listings above, only the last RDF triple actually contains the actual value of client connections, whereas the other three triples serve to describe the semantic context of that value and do not change that frequently. In these circumstances, we can distinguish between truly dynamic and relatively static values, and sending both types of triples at the same rate may result in an increased amount of redundant data being sent over the network. Both issues – namely, optimisation of the RDF sampling rates and minimisation of redundant data sent over the network are seen as potential directions for future research and are further discussed in Chapter 8.

6.2.2 RDF streaming and C-SPARQL querying engine

The next step of the EXCLAIM data flow is the C-SPARQL query engine, which is responsible for detecting potentially critical situations and passing them further to the OWL/SWRL reasoning engine for analysis. Picking up messages from the monitoring queue is performed using the listener class provided by the RabbitMQ client library. The listener is registered with the queue and is responsible for deserialising incoming string messages into RDF triple objects and pushing them further to the C-SPARQL streaming engine. An instance of the C-SPARQL engine is configured as shown in Listing 6.7 below.

Listing 6.7: Initialising the C-SPARQL engine and registering a stream.

```
CsparqlEngine engine = new CsparqlEngineImpl();
engine.initialize();
```

```
CsparqlStream csparqlStream = new CsparqlStream("http://seerc.org/rdf/
    stream/");
engine.registerStream(csparqlStream);
final Thread thread = new Thread(csparqlStream);
thread.start();
```

In order for the C-SPARQL engine to function, it is required to initiate a data stream and register a standing C-SPARQL query against this data stream, as illustrated in Listing 6.8. Using the `WHERE` clause, the standing C-SPARQL queries serve to fetch potentially critical values from the incoming data stream. From this perspective, they can be seen as a filtering element – that is, their main responsibility is to let 'noisy' data pass through, while only critical triples are detected and passed on to the OWL/SWRL reasoning engine. Once the query is registered against the stream, the associated listener (i.e., `streamFormatter`) will start triggering every time RDF triples observed on the stream within the specified window frame satisfy the `WHERE` condition.

Listing 6.8: Registering a listener with the C-SAPRQL stream.

```
RDFStreamFormatter streamFormatter = new RDFStreamFormatter(
"http://seerc.org/rdf/stream/");
engine.registerStream(criticalStreamFormatter);

String query = "REGISTER␣QUERY␣query␣SELECT␣?s␣?p␣?o␣FROM␣STREAM␣<http://
    seerc.org/rdf/stream/>␣[RANGE␣1m␣STEP␣1s]␣WHERE␣{?s␣?p␣?o}";

CsparqlQueryResultProxy proxy = engine.registerQuery(query);
proxy.addObserver(streamFormatter);
```

6.2.3 OWL/SWRL reasoning engine

The RDF triples fetched by the C-SPARQL queries are then dynamically added to the ABox of the CSO, and the reasoning process is initialised. Depending on the set of registered SWRL rules, the newly-populated individuals may be classified as critical situations, thereby calling for a corresponding reactive action. As explained in the next chapter, same individuals may be classified as critical (i.e., belonging to the class `CriticalSituation`) or not, depending on the current level of service criticality. The following three listings below illustrate i) how an ontology is loaded and initiated within the EXCLAIM framework (see Listing 6.9), ii) how new potentially critical instances are added (see Listing 6.10) and, finally, iii) how the Pellet reasoner, when queried, can infer if there are any instances of the class `CriticalSituation` (see Listing 6.11).

Listing 6.9: Loading and initiating the CSO.

```
OWLOntologyManager oManager = OWLManager.createOWLOntologyManager();
OWLOntologyIRIMapper iriMapper = new SimpleIRIMapper(IRI.create("http://
    seerc.org/ontology.owl"), IRI.create(ONTOLOGY_BASE_URL));
manager.addIRIMapper(iriMapper);
iriMapper = new SimpleIRIMapper(IRI.create("http://seerc.org/postgres-
    rules.owl"),IRI.create("https://www.dropbox.com/s/8qb2eb6ike29c30/
    postgres-rules.owl?dl=1"));
manager.addIRIMapper(iriMapper);
OWLOntology ontology = oManager.loadOntologyFromOntologyDocument(IRI.
    create(RULES_BASE_URL));
OWLReasonerFactory reasonerFactory = PelletReasonerFactory.getInstance();
OWLReasoner reasoner = reasonerFactory.createReasoner(ontology, new
    SimpleConfiguration());
```

Listing 6.10: Adding new individual assertions to the CSO.

```
OWLDataFactory factory = oManager.getOWLDataFactory();
PrefixOWLOntologyFormat pm = (PrefixOWLOntologyFormat) oManager.
    getOntologyFormat(ontology);
OWLObjectProperty objProperty = oManager.getObjectProperty(pred);
if (null != objProperty) {
        // Add object property assertion axiom
        OWLIndividual object = oManager.getIndividual(obj);
        oManager.addObjectPropertyAssertion(individual, objProperty,
            object);
} else {
        // Add data property assertion axiom
        OWLDataProperty dataProperty = oManager.getDataProperty(pred);
        if (null != dataProperty) {
        oManager.addDataPropertyAssertion(individual, dataProperty, obj);
}
```

Listing 6.11: Querying the reasoner whether there are critical instances.

```
Set<Node<OWLNamedIndividual>> nodes = oManager.getClassInstances("
    CriticalSituation", false);
if (null != nodes) {
        LOGGER.warn(nodes.size() + "␣CRITICAL␣SITUATIONS␣DETECTED␣");
        for (Node<OWLNamedIndividual> node : nodes) {
                Set<OWLClassAssertionAxiom> parents = oManager.
                    getSuperclass(node.getRepresentativeElement());
                StringBuilder sb = new StringBuilder("␣␣␣␣␣" + node.
                    toString() + "␣:");
                Iterator<OWLClassAssertionAxiom> iter = parents.iterator
                    ();
                while (iter.hasNext()) {
```

```
                        OWLClassAssertionAxiom axiom = iter.next();
                        sb.append("␣" + axiom.getClassExpression().
                            toString());
                }
                LOGGER.warn(sb.toString());
        }
}
```

Listing 6.11 also contains a place-holder, which can be used to trigger certain actions upon detection of critical situations. Currently, it simply prints out a warning message in the console, but any kind of user notification and response mechanism can be hooked up at this point. The EXCLAIM framework is seen as part of the larger MAPE-K functionality, which will enable complete self-managing functionality. Accordingly, corresponding planning and execution components will be notified of detected critical situations here as well.

6.3 Cloud Sensor Ontology

Representing the central Knowledge element of the MAPE-K reference model (see Figure 2.2), the CSO is also the core component of the EXCLAIM framework. Its vocabulary of terms is accessed and used at every step of the information processing workflow within the framework. Therefore, we dedicate a separate section with a goal to explain the CSO and its role in the monitoring and analysis process.

6.3.1 Design process and methodology

The CSO was developed using Protege IDE, which is nowadays the *de facto* editor for engineering OWL ontologies (Jain and Singh, 2013, Tudorache et al., 2008). It is a free, open-source, feature-rich ontology editing environment with full support for OWL 2, and integration of several Description Logic (DL) reasoners like HermiT and Pellet. The IDE serves to create and edit one or more ontologies in a single workspace via a customisable user interface (i.e., users can arrange instruments and panels according to their individual preferences). Supported refactoring operations include ontology merging, moving axioms between ontologies, renaming of multiple entities, and many others. Visualisation tools allow for interactive navigation through ontological class hierarchies (both explicit and inferred).[1] It also

[1]For example, class diagrams included later in this section were generated by the built-in tools of Protege.

supports debugging features – e.g., advanced explanation support aids in tracking down inconsistencies.

The principles underpinning the design and implementation of the CSO reflected existing best practices and recommendations as to how sensor-enabled domains should be modelled and structured; apart from the SSN ontology, which was the main point of reference and inspiration in our research, other important influencers were OntoSensor (Russomanno et al., 2005) and Ontonym (Stevenson et al., 2009). Moreover, when developing the CSO, we also relied on established ontology engineering principles (Gruber, 1995, Uschold and Gruninger, 1996), such as clarity, coherence, consistency, extensibility and adoption of naming conventions. The following aspects of the CSO developmnt process can be highlighted in this respect:

- The ontology design process has been a thorough and iterative process. To ensure that the resulting ontology adequately represents the cloud platfrom environment, it is essential that multiple target CAPs were investigated, so as to extract generic commonalities between them. More specifically, the overall duration of the ontology design process is approximately three years. At the initial stage of the prototype development, the EXCLAIM target cloud platform was VMWare CloudFoundry, and some experiments were conducted with Red Hat OpenShift, IBM Bluemix, and MS Azure, and the latest target platform has become Heroku. By identifying commonalities between these platforms it was possible to devise the upper level of the CSO, which is expected to represent common concepts of CAPs in a generic platform-independent manner.

- Ontologies are known to be shared and agreed between multiple people (Gruber, 1993), thus minimising the amount of potential bias caused by an individual's personal commitments. Accordingly, several researchers have contributed their knowledge to discussions leading to the development of the CSO. In this respect, the CSO can be seen as a result of collaborative efforts of researchers from South-East European Research Centre (SEERC), CITY College Computer Science Department, The Department of Computer Science of the University of Sheffield, and the 'RELATE ITN' [1] consortium.

When shifting focus from the conventional physical sensor devices of the Sensor Web domain to the 'logical software sensors' of CAPs, many of the concepts defined in existing sensor ontologies become irrelevant and may be omitted. Mainly,

[1] http://www.relate-itn.eu

these are the concepts related to the physical placement and environment of sensor devices. Additionally, since existing ontologies primarily target sensor observations, they do not include concepts related to situation assessment and adaptations, and this was another challenge for us when developing the CSO.

As we have previously explained, the EXCLAIM framework is envisaged to be part of the target cloud platform and act on its behalf when collecting metrics from the applications and services. Below, we explain how and why various stakeholders can access the CSO.

- CAP provider is the primary stake-holder responsible for maintaining and further developing the ontology (i.e., mainly – its lower level). The provider is expected to update the lower ontology with newly-added entities (i.e., services, applications, and platform components) and remove outdated entities concepts. This so-called 'zoo keeping' is not expected to be associated with frequent changes, as the core CSO is seen as a relatively static vocabulary.
- Third-party service providers are responsible for their respective linked extensions (as explained in Section 5.3.1). They are not allowed to interfere with the core ontology, but rather should design and maintain policies concerning their provisioned add-on services by extending the core CSO. Frequency of changes to linked extensions may vary, but typically is not expected to be high.
- CAP consumers are not expected to directly interfere with the ontology, which always remains at the back end of the EXCLAIM framework. As it frequently happens in the cloud paradigm, users are typically unaware of what happens behind the scenes – they only interact with the front-end Graphical User Interface (GUI) of the framework, and are unaware of specific detail its internals.

Logically, CSO can be divided into an upper (i.e., platform-independent) and a lower (i.e., platform-specific) levels. The former contains high-level concepts which are potentially reusable across multiple CAPs, whereas the latter contains domain-specific knowledge, such as actual cloud service names and their properties. Accordingly, as far as the principle of ontology completeness[1] is concerned, the upper-level CSO is seen as a rather static and complete model, whereas the lower level is expected to be more dynamic – that is, it is supposed to be extended

[1]Ontology completeness refers to an adopted and established practice to model a particular domain as completely as possible. Primarily, it refers to the 'horizontal' completeness – that is, including all possible concepts at the same hierarchical level, as opposed to the 'vertical' completeness, which refers to focussing on a single branch of an ontology.

and adjusted to capture concepts relevant to a specific CAP, its add-on services and internal organisation.

6.3.2 Upper level of the Cloud Sensor Ontology

The upper ontology includes 4 classes: `Sensor`, `Property`, `Situation`, and `ContextObject`, which have been identified as common across multiple CAPs:

- `Sensor` (see Figure 6.2) – this is the main class used to describe sensors within CAPs – that is, entities, which are expected to generate data for later monitoring and analysing. These concepts represent common elements of service-based cloud environments, are present in all surveyed CAPs. Instantiations of this class are used by triplification connectors to construct RDF streams. We can identify the following sensor entities, generic across various CAPs:
  - `Service` represents platform add-on services – main elemnts to be monitored in the context of the EXCLAIM framework.
  - `User` represents the CAP user, whose behaviour needs to be observed to perform data analysis.
  - `Application` represents user software deployed on the CAP and coupled with add-on services.
  - `Component` represents internal elements of services, applications or CAPs themselves.

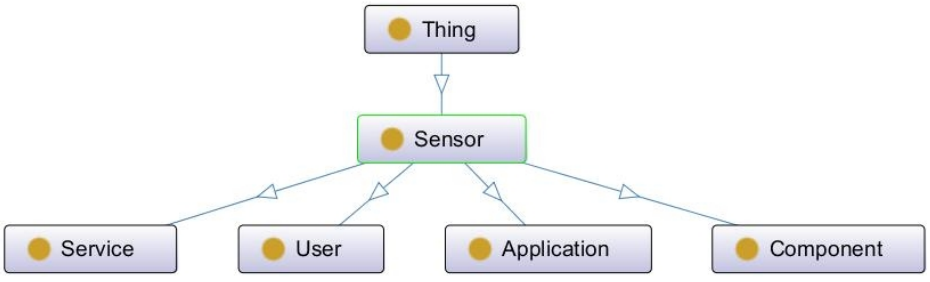

Figure 6.2: `Sensor` class is used to model entities whose certain aspects have to be measured and monitored.

- `Property` (see Figure 6.3) – this class describes various metrics of software sensors to be observed and interpreted by the EXCLAIM framework. Similar to the `Sensor` class, `Property` instances serve to construct RDF triples by

triplification connectors, which follow the intuitively comprehensible 'Sensor – Observes – Property' pattern adopted by the SSN, OntoSensor and Ontonym ontologies. This pattern facilitates conciseness and enables defining the upper concepts (i.e., `Sensor`, `hasProperty`, and `Property`) first, and then extending them with required subclasses and sub-properties, thus avoiding redundancy and repetitions. We can identify the following sensor properties, generic across various CAPs and add-on services:

- `Time` represents an exact point of time, when an event occured.
- `Duration` represents a certain amount of time between two events – e.g., between the moment when a process started and the moment when it terminated.
- `Size` represents a measurable amount of resources (e.g., size of a data base)
- `Number` represents a countable amount of resources (e.g., number of client connections to a service).
- `BooleanProperty` indicates whether a particular sensor entity has a certain feature or not – that is, a boolean value (e.g., a messaging queue is backed up).

The `Property` class is related to `Sensor` through the `hasProperty` object property, which is further sub-classed into `hasTime`, `hasDuration`, `hasSize`, `hasNumber`, and `hasBooleanProperty`.

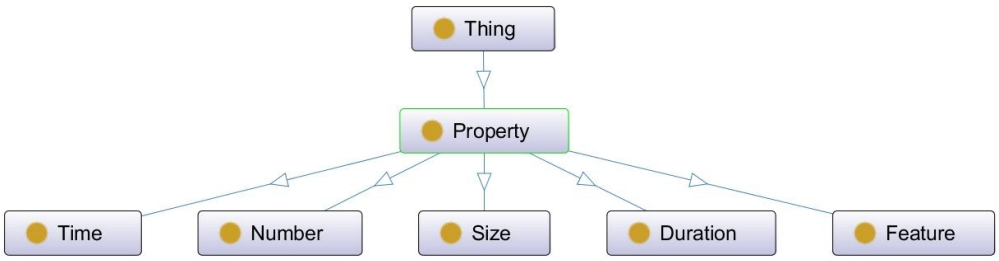

Figure 6.3: `Property` class represents characteristics of sensors which have to be measured and monitored.

- `ContextObject` (see Figure 6.4) – this class serves to model all other entities within CAPs, which represent the context of the managed element. Instances of this class never appear on RDF streams (i.e., are not used by triplification connectors), but are used to construct the static part of the analysis component – that is, SWRL policies and C-SPARQL queries (whenever there is a

need to match streaming data with static data, which may also be defined using the `ContextObject` class). Accordingly, `ContextObject` instances are primarily used by the C-SPARQL and OWL/SWRL engines, but may also act as sensors in their own right.[1] We can identify the following context objects, generic across various CAPs:

- `PlatformObject` represents an entity belonging to the cloud platform itself.
- `UserObject` represents an entity belonging to the CAP user.
- `ApplicationObject` represents an entity belonging to software deployed on the cloud platform.
- `ServiceObject` represents an entity belonging to add-on services.

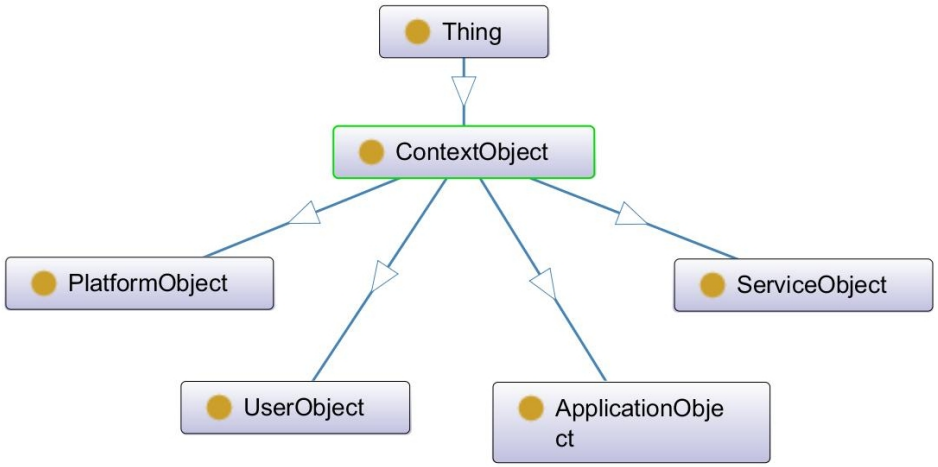

Figure 6.4: `ContextObject` class contains elements required to define the static knowledge.

- `Situation` (see Figure 6.5) – this class is intended to be used only by the OWL/SWRL reasoning engine to distinguish between ordinary (albeit *potentially* critical) situations and situations, which are indeed critical and require certain responsive actions to be taken. Accordingly two main sub-clases constitute this class: We can identify the following context objects, generic across various CAPs:
  - `OrdinarySituation` represents a situation, when collected observations are not critical and do not require responsive actions.

[1]OWL allows multiple inheretence, and thereby individuals may belong to `ContextObject` and `Sensor` classes.

– `CriticalSituation` represents a situation, which may threat the stability of the system or individual deployed applications, and thereofre has to be reported and acted upon.

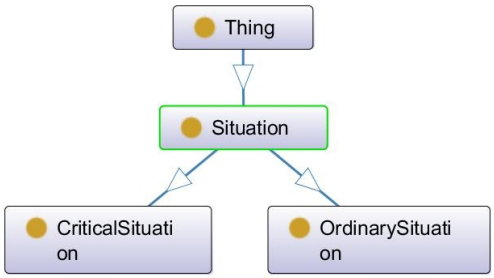

Figure 6.5: `Situation` class is used to classify currently observed situation and confirm that a potentially critical situation is indeed critical.

The described classes collectively constitute the upper-level CSO, which is supposed to be fairly static. Nevertheless, we also expect occasional changes and modifications – e.g., introduction of a cluster of add-on services by multiple CAPs. For example, nowadays there is an increasing demand for various IT services related to the emerging IoT, and CAPs are likely to start offering such services in the near future. As a result, a corresponding class will need to be added to the CSO by the CAP providers.

6.3.3 Lower level of the Cloud Sensor Ontology: Heroku-specific concepts

The lower level of the CSO represents platform-specific aspects of the target cloud environment, which are intrinsic to a particular CAP. In the context of the present research work we focus on Heroku, and this section briefs the reader on some of the add-on services offered by the Heroku marketplace – this material will help to understand the next chapter, where we present a case study focusing on the Heroku platform.

Please note, here we provide deliberately simplified class diagrams, which only include concepts relevant to the use case scenario to be explained in the next chapter. For the full diagram of the CSO, including the upper and lower levels, please refer to Appendix B.

- The `Sensor` class (see Figure 6.6) is extended with `HerokuService`, `HerokuDatabaseService`, and `HerokuPostgresService`. These classes

represent Heroku's all add-on services, storage services, and the PostgreSQL storage service respectively. The `Component` also includes classes `PlatformComponent` and `DatabaseServer`.

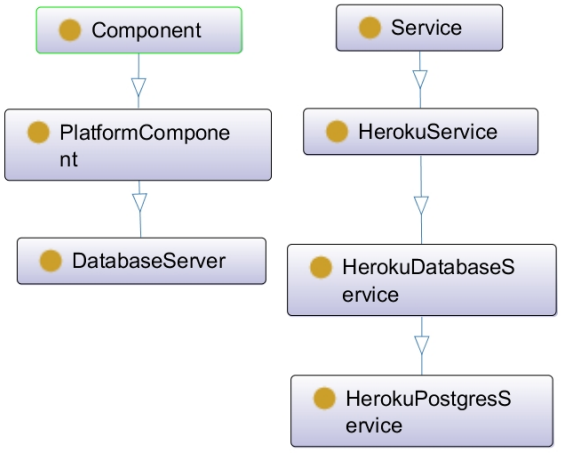

Figure 6.6: Heroku-specific concepts belonging to the `Sensor` class.

- The `Property` class (see Figure 6.7) is extended with the properties `NumberOfConnections` and `DatabaseSize`.

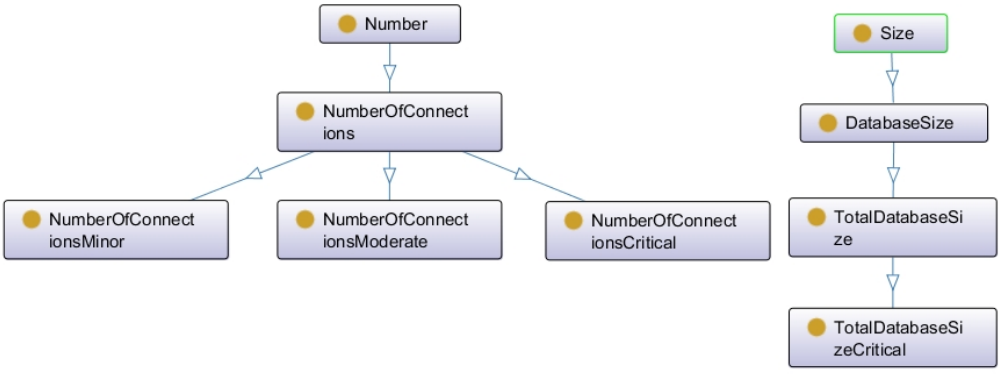

Figure 6.7: Heroku-specific concepts belonging to the `Property` class.

- The `ContextObject` class (see Figure 6.8) is extended with the `DatabaseServer` class, which is used to define the static background knowledge of the SWRL reasoner, as explained in the next Chapter.

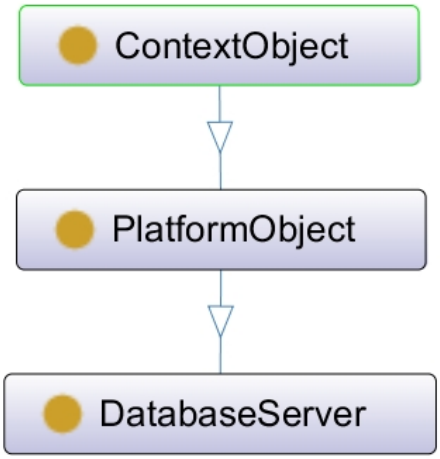

Figure 6.8: `ContextObject` class contains elements required to define the static knowledge.

- The `Situation` class (see Figure 6.9) is extended with two critical situations – i.e., `TotalDatabaseSizeCritical` and `NumberOfConnectionsCritical`, which serve to classify observations as critical.

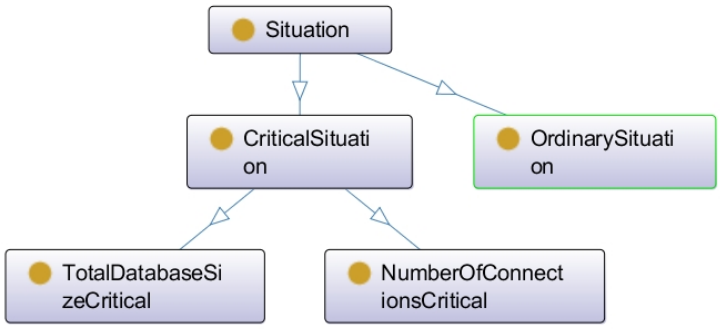

Figure 6.9: `Situation` class is used to classify currently observed situation and confirm that a potentially critical situation is indeed critical.

6.3.4 SWRL policies and linked extensions

Besides the core CSO, the knowledge base of the EXCLAIM framework is equipped with SWRL detection policies, which determine whether particular (potentially critical) observations represent an indeed critical situation. These policies typically concern individual add-on services, and are expected to be developed and maintained by service providers themselves. This separation of concerns be-

tween the CAP provider and third-party service providers is achieved by applying Linked Data principles and breaking down the set of policies into multiple distributed extensions deployed on the Web.

Typically, every linked policy is represented with three different SWRL rules, which correspond to the criticality level applied by the application owner, when profiling and deploying software on a CAP. That is, after the application has been deployed on the platform, the framework is able to parse the associated profile to decide which policies have to be applied in the current circumstances. This allows the framework to load only those SWRL rules, which are required in the given context, and avoid unnecessary invocations of the reasoner for a much larger set of rules.

Such a modular approach is underpinned by the core feature of Linked Data – unique identification of resources on the Web, which enables importing of distributed Web documents at run-time by crawling their URIs.

6.4 Opportunities and requirements for cross-platform deployment of the EXCLAIM framework

The EXCLAIM framework is designed and implemented to be re-usable and applicable across a wide range of service-based cloud platforms. As outlined in Chapter 2.2, among the whole PaaS market we have identified 24 CAPs, which are seen as a potential application scope for our proposed EXCLAIM framework. Therefore, the design principles underpinning the implementation of the EXCLAIM framework are based on the idea of making the framework as much platform-independent as possible, thus minimising (and, ideally, completely eliminating) the amount of code modifications, associated with the process of deploying the framework.

To support this claim of cross-platform applicability of the EXCLAIM framework we now describe and discuss the main steps required to be taken to deploy the framework on a new cloud platform. We remind the reader that in describing the EXCLAIM framework we take the perspective of the cloud platform provider and therefore assume being granted access to the standard metrics of the provisioned add-on services via API end-points.

One of the main advantages of the EXCLAIM framework is its non-intrusive approach to data collection. It targets at collecting data by means of already existing probes, and therefore does not require intrusive code modifications. It means that as long as the left side of the MAPE loop (Monitoring and Analysis) is concerned, EXCLAIM users (i.e., CAP providers) are exempted from a potentially effort-intensive routine of 'inserting' probes into the source code. Rather, the data

is extracted and collected using standard metrics across add-on services belonging to users' applications – access to these metrics is provisioned via the management API of particular add-on services.

What needs to be implemented, however, are the connectors – software components responsible for the triplification of the extracted raw values. At its current state, the EXCLAIM framework relies on hand-coded mappings between the raw data and its RDF representation using the CSO vocabulary. However, this challenge can be minimised – that is, the triplification connectors currently implemented for Heroku can be potentially re-used for similar add-on services existing in the target CAP. Furthermore, as the collection of already implemented connectors will grow, covering more and more CAPs and services, the task of implementing a connector will be replaced by the trivial task of picking the right connector from the already existing ones. To a certain extent, this process can be automated such that the CAP provider will only need to declaratively map between add-on service raw data value and the corresponding CSO terms.

Another software element of the current version of the EXCLAIM framework prototype, which might need to be changed, concerns the messaging queue component. At its current version, the framework uses the RabbitMQ add-on provisioned by the Heroku marketplace, it will still be able to use these queues even after it has been deployed on a new CAP. However, using a local messaging service is expected to minimise the unnecessary network latency, which might occur due to using an external (relatively remote) service. Accordingly, the target CAP provider might want to change the respective URLs pointing to a particular RabbitMQ host server.[1]

As far as the knowledge base (i.e., the CSO and SWRL rules) of the EXCLAIM framework is concerned in the context of cross-platform deployment, there are certain changes which need to be made. As we have previously seen, the CSO implements a two-tier architecture – i.e., the upper-level ontology is platform-independent and models the common generic features of CAPs, whereas the lower-level ontology further extends it with platform-specific concepts of a particular CAP. For example, the current version of the lower-level CSO describes the Heroku ecosystem.

Accordingly, the cross-platform deployment of the EXCLAIM framework only requires introducing changes to this lower level of the CSO so as to reflect the environment of the target CAP. In particular, the specific concepts belonging to the target CAP and describing its internal organisation (e.g., existing add-on services

[1]Please note again that no code modification is required, as the RabbitMQ client library is expected to work with any messaging queue service implementing the AMQP.

and applications, their properties, CAP components, context objects, etc.) have to be added. Consequently, newly-added ontological classes and properties will enable definition of RDF streams, C-SPARQL queries, and SWRL policies.

To sum up, the cross-platform deployment of the EXCLAIM framework is seen as a relatively straight-forward task, not involving considerable modifications and adjustments. The main challenge in these circumstances is to correctly capture the specific details of the target CAP and model them in the lower-level CSO, and further extend the knowledge base with corresponding RDF streams, C-SPARQL queries and SWRL policies. Based on these newly-added terms it is also required to implement corresponding triplification connectors, which would translate raw data to the RDF format.

6.5 Summary

In this chapter, we explained the actual implementation details of the EXCLAIM framework. The chapter builds upon the material of Chapter 5, and briefs the reader on technical aspects of the framework, which are also required to understand the case study and experiments to be descibed in the next chapter. More specifically, the chapter explained how the main three components of the framework – namely, the triplification engine, the C-SPARQL querying engine, and the OWL/SWRL reasoning engine – are implemented and utilise the CSO. A separate section is dedicated to the CSO, which describes its main concepts, constituting its upper and lower levels. In the next chapter, we will explain in more detail how the CSO is actually used within the CSO by demonstrating a case study focusing on the Heroku add-on services.

Chapter 7

Proof of concept: monitoring and analysis of Heroku add-on services with the EXCLAIM framework

This chapter considers a case study, which involves a Heroku-based application, which is coupled with 5 add-on services offered by this platform marketplace. The use case scenario s descibed from two perspectives of the key stake-holders - namely, the platfrom provider and the consumer. With the use case scenario explained, we will be able to describe each of the main three components of the framework – namely, the triplification, querying and reasoning engines. In all three of these components, a key role belongs to the CSO model, which provides vocabulary to define RDF triple streams, continuous SPARQL queries, and SWRL rules respectively. Accordingly, the use case demonstrates how raw data is first transformed into RDF, then queried with C-SPARQL queries, and eventually analysed by the OWL/SWRL reasoning engine.

To demonstrate the potential of the framework to scale, we also include a description of using an existing Big Data processing platform – i.e., IBM Streams – to create a parallelised deployment of the framework. The chapter presents experimental results of the framework performance with respect to the presented case study.

7.1 Case study: Destinator – an application for sharing destinations on Facebook

To describe the case, we first need to familiarise the reader with a cloud-hosted application, which served to conduct the case study and will help to understand how the EXCLAIM framework functions. The presented use case scenario will enable us to demonstrate the viability of the whole presented approach, run real-life experiments and benchmark the performance of the EXCLAIM framework.

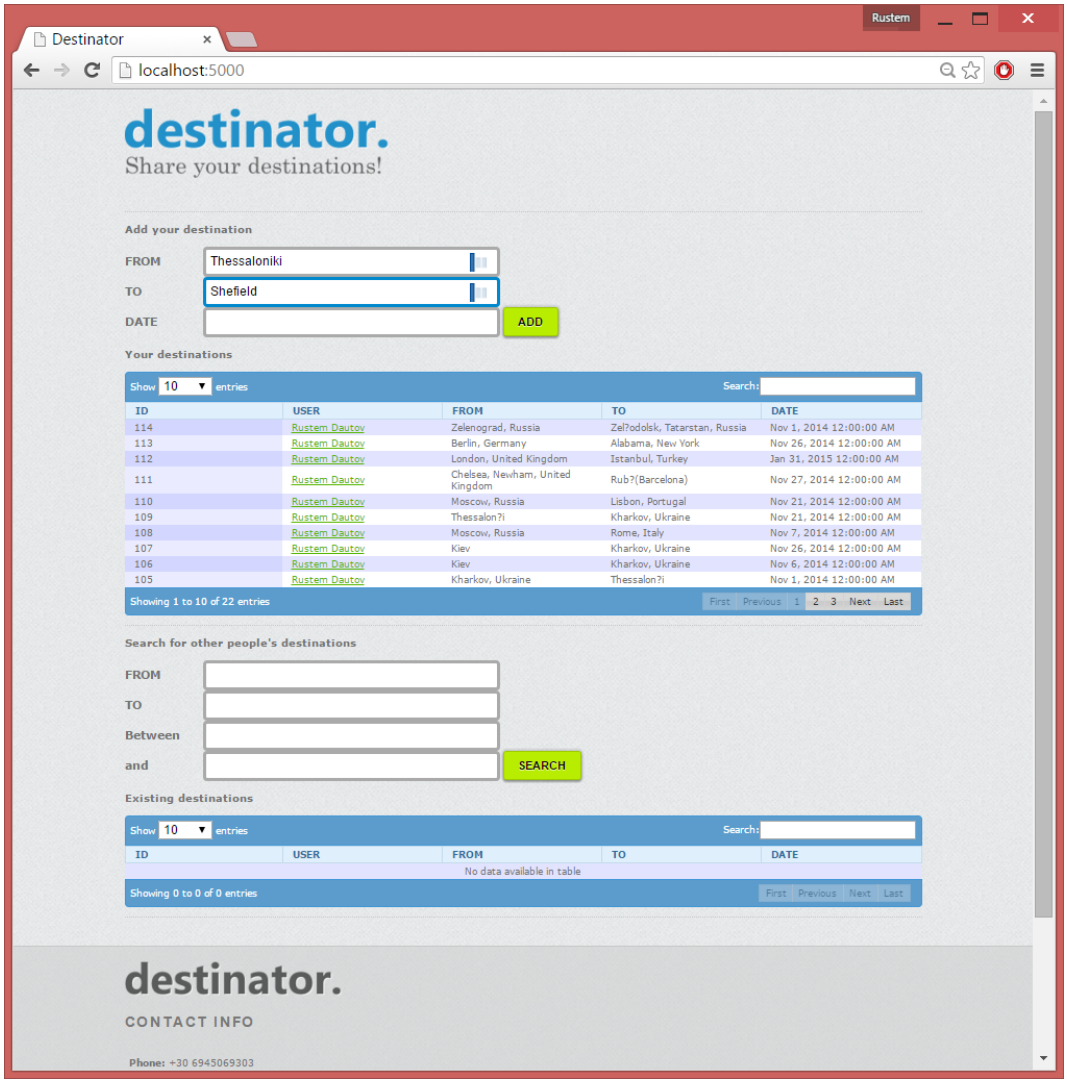

Figure 7.1: Destinator – a Heroku-based application for sharing destinations on Facebook.

For the requirements of this research work, we developed a Heroku applica-

tion for Facebook called Destinator[1] (see Figure 7.1). Facebook users can access the application as a Facebook canvas application[2] or directly on Heroku[3]. The idea behind it is simple – from time to time, different people ask through their social networks if someone is travelling from City A to City B in the next few days, weeks, or months. Typically, this happens when they want to send some personal items or documents, which they cannot post otherwise for various reasons. Posting such requests in social networks is, however, not very efficient, mainly because the chances to find someone travelling to the required destinations on the required dates among the limited audience of friends are not that high. Accordingly, the idea of Destinator is to let users a) share their destinations (i.e., where from, where to, and when), and b) search for people who might be travelling somewhere soon. For example, someone urgently needs to send documents to a friend of him/her from Thessaloniki to Athens tomorrow, but cannot travel him/herself, so he/she needs to search for someone else who is taking this trip tomorrow and has already shared this trip with Destinator. The application will search for people travelling from Thessaloniki to Athens, apply filtering based on the specified time frame and eventually fetch the list of potential travellers to the user. An attractive feature of Destinator is its ability to 'crawl' the so-called 'social graph' – a graph-like structure used to represent social connections within Facebook – in order to sort the result set of potential travellers with respect to the degree of social connection to the user. In other words, immediate friends will be displayed first, friends of friends will be displayed second, friends of friends of friends will be displayed third, and so on. This feature serves to help the user to approach a potential traveller when asking to deliver an item for him/her – that is, the user can, for example, ask his/her friend to approach the potential traveller on his/her behalf.

When implementing this application, one of the goals was to fully utilise the potential of Heroku and its add-on service marketplace. Our intention was to use existing services to save time and effort as much as possible. As a result, Destinator is connected with 5 add-on services provided by the Heroku marketplace (see Figure 7.2).

One of the main functionalities of Destinator is the ability to save and store users' destinations. The list of destinations is provided by Facebook API and displayed to the user as a drop-down list, after he/she types in first three characters

[1]Even though Destinator was developed and deployed from scratch in less than a month by efforts of just one software developer, it is nevertheless a publicly-available, working application and has the potential to be further enhanced and commercially exploited. The case study based on this application and presented in this chapter meets our requirements of evaluating the viability of the presented approach, as well as benchmarking its performance.

[2]`https://apps.facebook.com/209625285911924/`

[3]`http://destinator.herokuapp.com/`

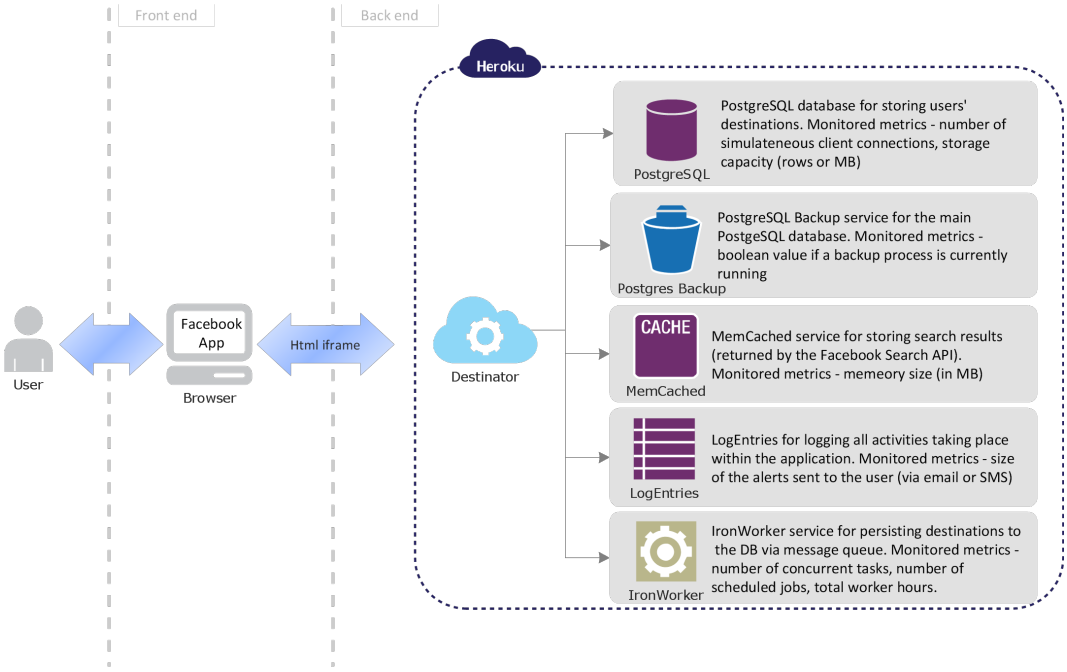

Figure 7.2: Architecture of the Destinator application.

of a place he will be travelling to/from. Accordingly, every time the user starts typing a destination name, a request is sent to Facebook API, which then replies with a list of destinations in JSON format. To minimise the number of network calls to Facebook API, we integrated a caching mechanism provided by Heroku's MemCached add-on service. It stores search results in the form of key-value string entries, where key is a search sequence, and value is a respective JSON response from Facebook API with a list of destinations, whose names start with the given search sequence. As a result, every time the user types in a new search string, Destinator first checks its local cache storage. In case the local cache storage does not contain this key yet, the request is forwarded to Facebook API, and the response is added to the local cache.

After users choose outbound and inbound destinations and a travelling date, this information is persisted to the PostgreSQL, which is one of the several enterprise-level storage services offered by Heroku. To avoid multiple simultaneous connections to the database, the storage functionality is decoupled from the main application by means of a messaging queue and worker processes attached to it. Destinator generates a payload with information about the destination (where to, where from, when, user metadata, etc.) and sends this task to the IronWorker messaging queue service. Worker processes then pick up incoming messages and persist data to the PostgreSQL service. Thus, persistence is done in a non-blocking

manner, and is not a bottleneck of the application. The PostgreSQL service is also coupled with the PGBackups service, which serves to back up all the data in the database on a daily basis.

The last but not the least connected add-on is the LogEntries service for logging. Since logging is nowadays recognised as an integral part of any industrial application, it was important to have this functionality so as to facilitate debugging, testing and notification routines.

Table 7.1 below summarises the above-mentioned Heroku add-on services. As it is seen from the table, each of the services is associated with certain measured metrics. These metrics represent resource quotas, which are provisioned to the user based on the subscription plan, and their consumption is measured by Heroku. For example, the PostgreSQL data storage service offers 14 subscription plans ranging from a completely free account (which is typically supposed to be used for trial and experimenting purposes) to a most expensive, full-blown subscription (targeted at creating large scale, enterprise-level application systems). Accordingly, under the free plan users can store up to 10,000 rows in the database, retain up to 2 backups, and establish up to 20 simultaneous client connections, whereas under the most expensive subscription plan users are provisioned with up to 1 TB of disk space and 120 GB of Random-Access Memory (RAM), up to 50 stored backups and up to 500 simultaneous client connections. Essentially, these subscription plans and associated resource constraints can be seen as SLAs between service subscribers and providers – for a given price paid by the customers, service providers are obliged to provide them with a corresponding service value. Similar pricing schemes and SLAs exist for all other add-on services offered by Heroku. Most of these metrics can be 'sensed' using standard APIs and entry points, and do not need instrumenting the source code.

Table 7.1: Five add-on services connected to Destinator, and their monitored aspects.

Add-on service	Monitored aspects and respective threshold values	Sensing mechanism

Heroku PostgreSQL[1] is an SQL database service run by Heroku that is provisioned and managed as an add-on. Heroku PostgreSQL is accessible from any language with a PostgreSQL driver including all languages and frameworks supported by Heroku: Java, Ruby, Python, Scala, Play, Node.js and Clojure. In addition to a variety of management commands available via the Heroku Command Line Interface (CLI), Heroku PostgreSQL features a web dashboard, the ability to create dataclips and several additional services on top of a fully managed database service.	Number of simultaneous client connections (up to 20) & storage capacity (up to 10,000 rows)	The data can be received by running an SQL query on an auxiliary table `pg_stat_activity`, which contains real time statistical information about the PostgreSQL database, including the current number of client connections and total storage utilisation of the database.
PGBackups[2] is Heroku's database backup solution for the PostgreSQL database. It can capture and restore backups from users' Heroku PostgreSQL databases for an application, import data from an existing database backup, and export backups for off-site storage.	Boolean value indicating whether the backup process is currently running & number of backups (up to 2 daily backups)	This data can be obtained by filtering the SQL query results regarding the current number of client connections to the database. The filtering is based on the field `application_name`. PGBackup connections are marked with 'pgbackups'.
IronWorker[3] is a massively scalable task queue service that makes it easy to offload front end tasks, run background jobs, and process many tasks at once. It can also function as a cron-in-the-cloud service, running tasks on a pre-defined schedule.	Number of concurrent tasks (up to 5 tasks) & number of scheduled jobs (up to 5 jobs) & total worker time (up to 10 worker hours)	This data can be obtained by executing a method provided by the IronWorker client library.

[1] `https://addons.heroku.com/heroku-postgresql/`
[2] `https://addons.heroku.com/pgbackups/`
[3] `https://addons.heroku.com/iron_worker/`

Memcached Cloud[1] is a fully-managed service for running Memcached in a reliable and fail-safe manner. Users' datasets are constantly replicated, so if a node fails, an auto-switchover mechanism guarantees data is served without interruption. Memcached Cloud provides various data persistence options, as well as remote backups for disaster recovery purposes. A Memcached bucket is created in seconds and from that moment on, all operations are fully-automated. The service completely frees developers from dealing with nodes, clusters, server lists, scaling and failure recovery, while guaranteeing no data loss.	Storage capacity (up to 30 megabyte (MB))	This data can be obtained by executing a method provided by the Memcached client library.
Logentries[2] is a simple, intelligent, intuitive and powerful solution for log management with out-of-the-box built-in Heroku specific alerting. Logentries presents log events from Heroku applications through an easy-to-use Web GUI and integrates directly with application performance monitoring tools like New Relic. Logentries also offers powerful graphic dashboard support for quick and easy visual representation of logged data.	Total size of email alerts and notifications sent to users (up to 5 GB per month)	This data can be obtained by executing a method provided by the Logentries client library.

Heroku itself measures and bills its customers for using add-on services. However, customers are not notified in advance when the resource consumption is reaching 'danger levels' – for example, too many simultaneous client connections to the PostgreSQL service. This can result in further connection requests being unexpectedly rejected. Accordingly, our goal in this case study was to equip services with sensing capabilities, so that we, as application providers, can be notified in advance whenever a critical threshold is approaching, and thereby be enabled to take appropriate pre-emptive actions – for example, by closing down low-priority idle connections or by automatically upgrading the Heroku subscription plan.

[1] `https://addons.heroku.com/memcachedcloud`
[2] `https://addons.heroku.com/logentries/`

7.2 Demonstrating monitoring and analysis capabilities

Having introduced and explained Destinator – the target Heroku-based application to be monitored by the EXCLAIM framework – we now proceed with a case study, aiming to demonstrate the viability of the approach and to showcase multi-faceted advantages of the proposed approach. The demonstration is structured around the two main roles (i.e., perspectives) concerned with monitoring and management of service-based applications deployed on the cloud platform.

7.2.1 Cloud Application Platform provider's perspective

Primarily, the EXCLAIM framework is intended to support and enhance the CAP provider's capabilities for platform governance, and therefore the provider is seen as the main stakeholder. We have already raised several issues the CAP provider might be concerned with. Among other things, one of the key challenges in the presence of numerous virtual tenants simultaneously accessing a common shared pool of cloud resources is to ensure that the amount of the underlying physical resources is sufficient. This underpins stable and undisrupted operation of the cloud services.

Given the motivation of the provider to minimise the amount of idle physical resources (for economic and ecological reasons), it is typically expected that physical servers are turned on and off dynamically in response to the ever-changing demands of the cloud platform consumers. In these circumstances, it is crucial for the cloud provider to be provisioned with a most recent overview of the overall resource consumption resulting from resource consumption of individual tenants.

More specifically, let us focus on Heroku's PostgreSQL data storage service, used by Destinator. When subscribing to this service, each subscriber is allocated a personal virtual space on a physical database server. Heroku aims at reducing operational costs by minimising the total number of physical database server instances running at the same time. This is typically achieved by increasing the 'tenant density' (i.e., number of virtual tenants on each of the physical servers), and, following the principle of elasticity, launching additional database servers only in case the existing capacity is not enough. Accordingly, the CAP provider needs to measure disk space utilisation of individual virtual tenants with respect to the total available physical space in order to launch additional server if needed or 'merge' several already running, but idle servers.

First, the framework needs to measure disk space utilised by an individual instance of the PostgreSQL service couple with Destinator. The next step is to sum up the collected individual values to get an overall occupied amount of Post-

greSQL disk space by Destinator. Finally, this value needs to be evaluated against the rest of the disk space occupied by other instances of PostgreSQL connected with other applications (i.e., not Destinator) on Heroku. Thus, the framework will be in a position to decide whether there is still enough space for the PostgreSQL service or not.

Below, we illustrate how this kind of use case scenarios is handled by the EXCLAIM framework to support cloud platform self-governance. The explanation is aligned with the three main steps, through which raw collected data is transformed as it passes through the EXCLAIM framework.

Triplification

The code snippet below (Listing 7.1) represents a stream, associated with the particular user instance of Destinator, to which all related RDF triples are sent. The snippet demonstrates how the disk space occupied by a PostgreSQL instance `postgres-service-10` changes with the time from 7,000 rows to 9,000 rows.

Listing 7.1: RDF triples on the stream indicate database space, occupied by an individual service.

```
cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasDatabaseSize cso:database-size-122
cso:database-size-122 rdf:type cso:DatabaseSize
cso:database-size-122 cso:hasValue "7000"^xsd:int

cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasDatabaseSize cso:database-size-122
cso:database-size-122 rdf:type cso:DatabaseSize
cso:database-size-122 cso:hasValue "7500"^xsd:int

cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasDatabaseSize cso:database-size-122
cso:database-size-122 rdf:type cso:DatabaseSize
cso:database-size-122 cso:hasValue "8000"^xsd:int

cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasDatabaseSize cso:database-size-122
cso:database-size-122 rdf:type cso:DatabaseSize
cso:database-size-122 cso:hasValue "8500"^xsd:int

cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasDatabaseSize cso:database-size-122
cso:database-size-122 rdf:type cso:DatabaseSize
cso:database-size-122 cso:hasValue "9000"^xsd:int
```

C-SPARQL querying

The next step is extract triples representing disk space occupation from the RDF stream. This task can be achieved by registering the following C-SPARQL query (Listing 7.2). The query only fetches real values, which are greater than zero, thus filtering out values coming from applications, which are staying idle, and thereofre should not be taken into consideration. The resuts of the query are sent forward to `CriticalStream`, which serves to deliver potentially critical triples to the OWL/SWRL reasoning engine.

Listing 7.2: A C-SPARQL query fetching the current value of the disk space utilisation by the HerokuPostgres service.

```
REGISTER STREAM CriticalStream
AS PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cso: <http://seerc.org/ontology.owl#>
CONSTRUCT { ?service rdf:type cso:HerokuPostgresService .
            ?service cso:hasDatabaseSize ?size .
            ?size rdf:type cso:DatabaseSize .
            ?size cso:hasValue ?value }
FROM STREAM <http://seerc.org/rdf/destinator-stream/> [RANGE 1m STEP 1s]
WHERE { ?service rdf:type cso:HerokuPostgresService .
        ?service cso:hasDatabaseSize ?size .
        ?size rdf:type cso:DatabaseSize .
        ?size cso:hasValue ?value .
        ?value > 0 }
        AGGREGATE { (?totalvalue, SUM, {?value})
        FILTER (?totalvalue > 1000000) }
```

OWL/SWRL reasoning

At the previous step, we assumed that all instances of Destinator (and therefore all instances of the HerokuPostgres service associated with it) send its measured values to a dedicated data stream (i.e., in the examples above – `http://seerc.org/-rdf/destinator-stream/`). In these circumstances, an individual C-SPARQL query registered with this stream can only process values coming from this particular application and is incapable of providing a more global view on the overall disk space consumption by the HerokuPostgres service connected to other Heroku applications. In these circumstances, it is important to gather information coming from various data streams in one place, and see if the overall utilisation is still within limits. The SWRL reasoning engine is intended to play the role of such a central component. In this sense, it is akin to a central component of a sensor network, which collects data coming from distributed sources.

All values appearing on `CriticalStream` are added to the CSO by the EXCLAIM framework, and the reasoning process is initiated. Listing 7.3 depicts an SWRL policy which sums up the total amount of disk space occupied (for demonstration purposes we assume the presence of only five applications using the HerokuPostgres service).[1]

Listing 7.3: New SWRL rule detecting situations when the overall database size dedicated to the HerokuPostgres service (shared by five different applications) is critical.

```
HerokuPostgresService(?ser1) ^ HerokuPostgresService(?ser2) ^
    HerokuPostgresService(?ser3) ^ HerokuPostgresService(?ser4) ^
    HerokuPostgresService(?ser5) ^
DatabaseSize(?size1) ^ DatabaseSize(?size2) ^ DatabaseSize(?size3) ^
    DatabaseSize(?size4) ^ DatabaseSize(?size5) ^
hasDatabasSize(?ser1,?size1) ^ hasDatabasSize(?ser1,?size1) ^
    hasDatabasSize(?ser1,?size1) ^ hasDatabasSize(?ser1,?size1) ^
    hasDatabasSize(?ser1,?size1) ^
hasValue(?size1, ?v1) ^ hasValue(?size2, ?v2) ^ hasValue(?size3, ?v3) ^
    hasValue(?size4, ?v4) ^ hasValue(?size5, ?v5) ^
swrlb:greaterThan(1000000000, op:numeric-add(?v1, ?v2, ?v3, ?v4, ?v5)) ->
CriticalTotalDatabaseSize(?ser1) ^ CriticalTotalDatabaseSize(?ser2) ^
    CriticalTotalDatabaseSize(?ser3) ^ CriticalTotalDatabaseSize(?ser4) ^
    CriticalTotalDatabaseSize(?ser5)
```

With SWRL it is also possible to combine dynamic values coming from the monitoring stream with a more static facts. We can modify the previous SWRL example with an assumption that the total amount of disk space occupied by HerokuPostgres services being greater than the dangerous threshold is only critical if there are no additional physical servers available to the CAP provider. Otherwise, it would be possible to elastically provision extra storage using the additional server. To represent this kind of policy with SWRL we assume the presence of a `ContextObject` instance `PhysicalServer` in the CSO – this instance does not come from the monitored stream, but is part of the background static knowledge. Accordingly, the modified SWRL rules is depicted in Listing 7.4.

Listing 7.4: A critical situation is detected only when there does not exist an additonal server.

```
HerokuPostgresService(?ser1) ^ HerokuPostgresService(?ser2) ^
    HerokuPostgresService(?ser3) ^ HerokuPostgresService(?ser4) ^
    HerokuPostgresService(?ser5) ^
```

[1]Please note how we make use of the SWRL built-in operators `op:numeric-add` and `swrlb:greaterThan`, which are used to calculate a sum of several values and check if it is greater than some other value respectively. The number 1,000,000,000 indicatively represents a dangerous threshold for the HerokuPostgres service.

```
DatabaseSize(?size1) ^ DatabaseSize(?size2) ^ DatabaseSize(?size3) ^
    DatabaseSize(?size4) ^ DatabaseSize(?size5) ^
hasDatabasSize(?ser1,?size1) ^ hasDatabasSize(?ser1,?size1) ^
    hasDatabasSize(?ser1,?size1) ^ hasDatabasSize(?ser1,?size1) ^
    hasDatabasSize(?ser1,?size1) ^
hasValue(?size1, ?v1) ^ hasValue(?size2, ?v2) ^ hasValue(?size3, ?v3) ^
    hasValue(?size4, ?v4) ^ hasValue(?size5, ?v5) ^
swrlb:greaterThan(1000000000, op:numeric-add(?v1, ?v2, ?v3, ?v4, ?v5)) ^
DatabaseServer(?dbs) ^ exists(?dbs, false) ->
CriticalTotalDatabaseSize(?ser1) ^ CriticalTotalDatabaseSize(?ser2) ^
    CriticalTotalDatabaseSize(?ser3) ^ CriticalTotalDatabaseSize(?ser4) ^
    CriticalTotalDatabaseSize(?ser5)
```

7.2.2 Cloud Application Platform consumer's perspective

The second part of the case study is intended to demonstrate the consumer's perspective on the monitoring process and how CAP users can benefit from using the EXCLAIM framework.

With several samples, we will illustrate how the framework is able to detect an excessive number of client connections to Heroku's PostgreSQL database service.

Heroku's pricing model offers customers a range of subscription plans, each associated with certain service levels – i.e., amount of metered and billed resources available to the user. In particular, a typical metric relating to data storage services is the number of simultaneous client connections. However, customers are not currently notified in advance when the number of active connections is reaching 'danger levels', and this can result in further connection requests being unexpectedly rejected. Accordingly, our goal in this case study was to equip data storage services with sensing capabilities, so that application providers can be notified in advance whenever a threshold is approaching, allowing them to take appropriate pre-emptive actions – for example, by closing down low-priority connections or by automatically upgrading their subscription plan.

Using our framework, we manually annotated sensory data (in this case, the current pool of client connections and the current state of the database backup process) with semantic descriptions defined in the CSO to generate a homogeneous data representation, and then streamed these RDF values to the C-SPARQL querying engine.[1] The maximum number of client connections for the initial subscription plan is limited to 20, and our goal was to detect situations when this number increases and approaches the danger level.

[1] To extract these metrics from the PostgreSQL service we relied on standard mechanisms offered by this database. See Table 7.1 for details of how this can be done.

Services and SBAs can be classified with respect to their criticality level (i.e., minor, moderate, and critical). Destinator, being a data-intensive application heavily depending on the underlying data storage service PostgreSQL, is therefore can be classified as critical in terms of its data storage service, and corresponding diagnosis policies have to be applied in this respect. Accordingly, as the critical level threshold we defined the value of database client connections to be 15 – it means that the EXCLAIM framework has to be particularly sensitive to Destinator, and detect potential critical situations well before the number of connections reaches 20 so as to notify the customer in advance and ensure application stability.

The following RDF stream in Listing 7.5 represents a situation when the number of client connections increased from 12 to 15, and no backup process is running – this is important because the backup process establishes two client connections to the database, but typically lasts for less than a minute, and therefore should not be considered as a threat.

Listing 7.5: RDF triples on the stream indicate an increase in the number of client connections to the PostgreSQL service from 12 to 15 connections with no backup process running.

```
cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasNumberOfConnections cso:number-of-
    connections-122
cso:number-of-connections-122 rdf:type cso:NumberOfConnections
cso:number-of-connections-122 cso:hasValue "12"^xsd:int

cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasNumberOfConnections cso:number-of-
    connections-122
cso:number-of-connections-122 rdf:type cso:NumberOfConnections
cso:number-of-connections-122 cso:hasValue "13"^^xsd:int

cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasNumberOfConnections cso:number-of-
    connections-122
cso:number-of-connections-122 rdf:type cso:NumberOfConnections
cso:number-of-connections-122 cso:hasValue "14"^^xsd:int

cso:backup-service-8 rdf:type cso:BackupService
cso:backup-service-8 cso:accesses cso:postgres-service-10
cso:backup-service-8 cso:isActive "false"^^xsd:boolean

cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasNumberOfConnections cso:number-of-
    connections-122
cso:number-of-connections-122 rdf:type cso:NumberOfConnections
```

```
cso:number-of-connections-122 cso:hasValue "15"^^xsd:int
```

In order to assess the current situation and detect violations we registered a set of standing C-SPARQL queries, which are evaluated 10 times per second (i.e., queries are evaluated every 100 ms). Listings 7.6, 7.7, and 7.8 represent the critical-level, moderate-level and minor-level criticality queries respectively. It needs to be noted that these three queries represent the previously-explained concept of application profiling (see Section 5.3) – that is, when users deploy their software, they can specify how sensitive they are in terms of particular add-on services. Such an application profile allows the framework to apply a differentiated approach to problem detection – i.e., in some circumstances a potentially critical situation can be tolerated, whereas in some others (more sensitive) occasions it has to be treated carefully.

As it is seen from the listings, the conditions specified in the `WHERE` clauses are mutually exclusive, which means that only one query will trigger at a time. The `WHERE` clause specifies the actual danger threshold for the PostgreSQL service's client connections, and in the considered scenario it will trigger whenever the number of client connections during the previous minute reaches the threshold of 15 at least once, provided there is no backup process running – that is, there are indeed 15 connected clients, and there is a potential threat to the application stability.

Listing 7.6: Detecting number of client connections at the critical level.

```
REGISTER STREAM CriticalStream
AS PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cso: <http://seerc.org/ontology.owl#>
CONSTRUCT { ?service rdf:type cso:HerokuPostgresService .
        ?service cso:hasNumberOfConnectionsCritical
        "true^^http://www.w3.org/2001/XMLSchema#boolean" }
FROM STREAM <http://seerc.org/rdf/stream/> [RANGE 1m STEP 100ms]
WHERE { ?service rdf:type cso:HerokuPostgresService .
        ?service cso:hasNumberOfConnections ?numConn .
        ?numConn rdf:type cso:NumberOfConnections .
        ?numConn cso:hasValue ?v .
        ?bservice rdf:type cso:BackupService .
        ?bservice cso:accesses ?service .
        ?bservice cso:isActive "false^^http://www.w3.org/2001/XMLSchema#
            boolean"
        FILTER ( ?v >= 15 && ?v < 17) }
```

Listing 7.7: Detecting number of client connections at the moderate level.

```
REGISTER STREAM CriticalStream
```

```
AS PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cso: <http://seerc.org/ontology.owl#>
CONSTRUCT { ?service rdf:type cso:HerokuPostgresService .
        ?service cso:hasNumberOfConnectionsModerate
        "true^^http://www.w3.org/2001/XMLSchema#boolean" }
FROM STREAM <http://seerc.org/rdf/stream/> [RANGE 1m STEP 100ms]
WHERE { ?service rdf:type cso:HerokuPostgresService .
        ?service cso:hasNumberOfConnections ?numConn .
        ?numConn rdf:type cso:NumberOfConnections .
        ?numConn cso:hasValue ?v .?bservice rdf:type cso:BackupService .
        ?bservice cso:accesses ?service .
        ?bservice cso:isActive "false^^http://www.w3.org/2001/XMLSchema#
            boolean"
        FILTER ( ?v >= 17 && ?v < 19) }
```

Listing 7.8: Detecting number of client connections at the minor level.

```
REGISTER STREAM CriticalStream
AS PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX cso: <http://seerc.org/ontology.owl#>
CONSTRUCT { ?service rdf:type cso:HerokuPostgresService .
        ?service cso:hasNumberOfConnectionsMinor
        "true^^http://www.w3.org/2001/XMLSchema#boolean" }
FROM STREAM <http://seerc.org/rdf/stream/> [RANGE 1m STEP 100ms]
WHERE { ?service rdf:type cso:HerokuPostgresService .
        ?service cso:hasNumberOfConnections ?numConn .
        ?numConn rdf:type cso:NumberOfConnections .
        ?numConn cso:hasValue ?v .
        ?bservice rdf:type cso:BackupService .
        ?bservice cso:accesses ?service .
        ?bservice cso:isActive "false^^http://www.w3.org/2001/XMLSchema#
            boolean"
        FILTER ( ?v >= 19) }
```

The `REGISTER` and `CONSTRUCT` clauses of the presented C-SPARQL queries serve to define a new RDF stream called `CriticalStream`, which only contains critical observations. Once any new triples appear on the stream, they are immediately added to the ABox of the CSO – that is, the ontology is populated with respective RDF triples. In our scenario, the CSO is populated with the following two triples defined in Listing 7.9:

Listing 7.9: The CSO is populated with RDF triples representing a critical situation.

```
cso:postgres-service-10 rdf:type cso: HerokuPostgresService
cso:postgres-service-10 cso:hasNumberOfConnectionsCritical "true"^^http:
    //www.w3.org/2001/XMLSchema#boolean
```

After they have been added to the ontology, the reasoning process is initiated. It is the most computationally-intensive part of the monitoring and analysis procedure, and we support this process by minimising the TBox of the CSO, and thus minimising the number of logical axioms with respect to which the reasoner evaluates the newly-added triples. Listing 7.10 contains SWRL rules, which can be potentially applied in this respect so as to classify the observed situation as critical or not.

Listing 7.10: Only one of the three SWRL rules, determining whether an observed situation is indeed critical or not, is added to the TBox of the knowledge base.

```
HerokuPostgresService(?s), hasNumberOfConnectionsMinor(?s, true) ->
    CriticalNumberOfConnections(?s)
HerokuPostgresService(?s), hasNumberOfConnectionsModerate(?s, true) ->
    CriticalNumberOfConnections(?s)
HerokuPostgresService(?s), hasNumberOfConnectionsCritical(?s, true) ->
    CriticalNumberOfConnections(?s)
```

This is where the two framework enhancements, described in Sections 5.3.1 and 5.3.2, again come into play. Since we classified Destinator as a data-intensive application demanding for critical-level policies to be applied, the knowledge base (i.e., the TBox) only includes the third SWRL rule from Listing 7.10. This rule is uploaded dynamically at run-time from a remote location (where it is maintained by the respective service provider) by following its URI and importing the fetched document into memory of the reasoning engine, as prescibed by the Linked Data principles. As a result, the monitored `HerokuPostgresService` instance is evaluated against a limited, yet targeted policies, and classified as belonging to the `CriticalNumberOfClientConnections` class.

The last step in this analysis process is to see if there are any instances of the class `CriticalSituation`. As it is illustrated in Figure 6.5, the class `CriticalNumberOfClientConnections` is a subclass of the class `CriticalSituation`, and the reasoner, resolving this subclass dependency, marks `HerokuPostgresService` as a critical situation. As a result, the EXCLAIM framework notifies the user of a detected critical situation.

It is worth noting that application profiling and enforcing of different-level policies only takes place at the very last step of OWL/SWRL reasoning. The C-SPARQL queries might have detected all three cases, when the number of client connections is gradually increasing from minor to moderate, and, eventually, to critical level (i.e., from 12 to 19, thus satisfying the `WHERE` conditions of all three queries in Listings 7.6, 7.7, and 7.8), and corresponding RDF triples are sent to the critical stream. However, as long as there are no SWRL rules added to the know-

ledge base, these values will not be marked as critical situations by the reasoner, which will only rely on the default, core ontology when resolving dependencies.

To justify the use of SWRL as a language for defining policies, we now demonstrate how its reasoning support can benefit the detection process. Listing 7.10 contains rules, which can only apply to a single database service available through Heroku marketplace – that is, HerokuPostgres. As indicated by Figure 8.8, it is a child of a higher-level class `DatabaseService` together with several other services. Accordingly, it is possible to rewrite the policies to make them more generic and applicable across all database services. By doing so, we benefit from SWRL's capabilities to resolve sub-class relationships, and minimise the amount of redundant policies. Listing 7.11 contains these generic rules:

Listing 7.11: Generic rules apply to all sub-classes of the class DatabaseService.

```
DatabaseService(?s), hasNumberOfConnectionsMinor(?s, true) ->
    CriticalNumberOfConnections(?s)
DatabaseService(?s), hasNumberOfConnectionsModerate(?s, true) ->
    CriticalNumberOfConnections(?s)
DatabaseService(?s), hasNumberOfConnectionsCritical(?s, true) ->
    CriticalNumberOfConnections(?s)
```

7.3 Deployment on IBM Streams

RDF stream processing in general and one of its existing implementations (i.e., C-SPARQL) in particular, at their current state suffer from performance and scalability issues – two aspects which can make our EXCLAIM framework potentially incapable of monitoring and analysing large data volumes within CAPs, given the fact that with our framework we aim to address real-world enterprise-level CAPs, which can potentially generate thousands and even millions of RDF triples per second. In our view, these amounts are large enough to be considered as Big Data not only because of the volume aspect, but also because of the existing velocity, variety and veracity of these datasets.

In these circumstances, a possible solution to overcome the problem is to parallelise processing tasks across several instances of the framework by fragmenting incoming data streams into sub-streams, so that each instance only deals with a separate subset of incoming values. To demonstrate this and prove out concepts, we required an existing infrastructure, which would allow us to implement such a parallel deployment with least possible refactoring and reconfiguration efforts. Given this, we were motivated to utilise an existing technological solution from the Big Data processing domain, which would exhibit the following characteristics:

- Minimal effort to integrate with our framework
- Support for processing streamed data
- Support for data stream fragmentation and task parallelisation
- Enough capacity to address the Big Data challenges of CAPs

As a result, among other alternatives IBM Streams was chosen as the target platform to implement a parallelised deployment of the EXCLAIM framework. Detailed description of this platform for processing streaming Big Data can be found in Section 3.1.1.

In the rest of this chapter, we will demonstrate and explain the parallelised deployment of the framework on top of the IBM Streams with the same use case scenario, which was presented in the previous chapter. We first explain the fragmentation logic, which we applied to partition RDF data on the stream, and then continue with a number of experiments. To a great extent, these experiments are similar to the ones explained in the previous chapter, and primarily serve to demonstrate an increase in performance when running the parallel deployment of the framework as opposed to the initial, pipelined deployment.

We also want to bring to the reader's attention that the primary goal of demonstrating the IBM Streams deployment is to show that the emerging Big Data issue can be successfully addressed, and to do so, the EXCLAIM framework has the capacity to be accordingly extended and integrated with existing solutions. Even though the experimental results show an increase in performance, our goal here is not to design and implement a novel efficient algorithm for data stream fragmentation and parallel processing. Rather, our intention is to demonstrate that it is possible in principle.

7.3.1 Stream parallelisation

So far, we focused on one particular service connected to the user's application – namely, the PostgreSQL data storage service. To demonstrate the potential of applying Big Data parallelisation techniques, we will take into account all five add-on services Destinator is connected to.

For the sake of demonstrating the benefits, we will apply a simple fragmentation logic, based on separating the main stream with RDF triples coming from different add-on services into several sub-streams, so that each of them only contains RDF triples originating from one particular service. Then, we will launch several identical instances of the EXCLAIM framework and attach them to the

resulting sub-streams. Eventually, we aimed at creating a distributed architecture with several replicated frameworks, each of which would only deal with data coming from a single add-on service.

With IBM Streams, which allows developing custom Java operators, we had to implement three main operators, which are depicted in the screenshot below (see Figure 7.3).

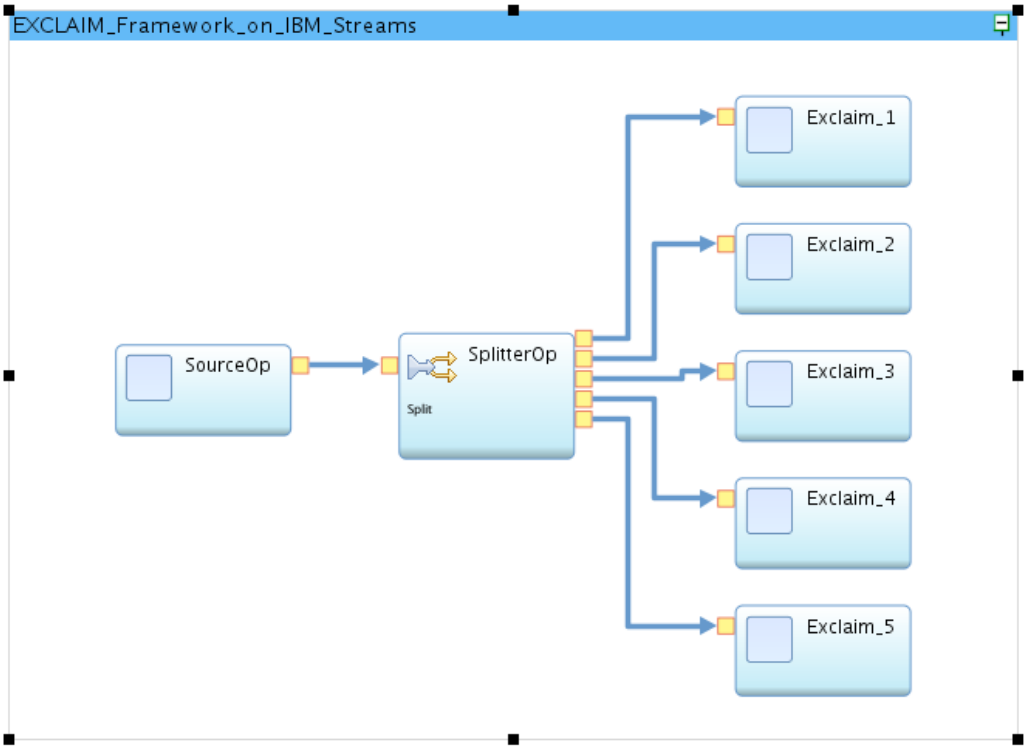

Figure 7.3: Parallel architecture with the main RDF stream split into five separate sub-streams, each of which is processed independently by several instances of the EXCLAIM framework.

- `SourceOp`: this operator acts as an entry point for RDF triples to be processed by the EXCLAIM framework. It is a data source operator, which picks up monitored RDF triples from the RabbitMQ monitoring queue, and then forwards them to the splitter operator.

- `SplitterOp`: this operator is responsible for the actual fragmentation of incoming data. Depending on the add-on service generating monitored values, the splitter directs data to a corresponding output port. Since in the considered case study, Destinator is coupled with 5 add-on services, the splitter has 5 respective output ports.

- `Exclaim_{1-5}`: this operator is essentially a Java wrapper for the EXCLAIM framework. Since it is called and executed from within the Streams platform, it does not have a GUI and a management console; therefore, all the configurable parameters are predefined. In the considered use case, there exist 5 identical instances of the `Exclaim` operator, each of which is attached to one of the output ports of the splitter. Thus, we achieve an architecture, in which each instance of the EXCLAIM framework (i.e., `Exclaim_{1-5}`) only deals with RDF triples belonging to a particular add-on service. Simply put, we minimise the number of incoming triples on each sub-stream, and thus achieve better performance.

To a certain extent, the splitter operator performs routing and filtering functions, and thus can be seen as an implementation of a routing node, existing in physical sensor networks. Such an intermediate node in the context of the EXCLAIM framework is responsible for:

- transferring monitored values from physical (e.g., server, data centre), virtual (e.g., application container, virtual machine), or logical (e.g., application system, database) components of the monitored platform to a corresponding processing location;

- performing initial processing of the incoming values – that is, by filtering and aggregating monitored values it is possible to offload some of the computational tasks from the central monitoring component (which otherwise may become a bottleneck of the whole system), and make the whole framework more scalable.

Unlike static data fragmentation, where the set of values is finite, partitioning of streamed data, due to its unbounded nature and unpredictable rate, is associated with a risk of splitting semantically connected RDF triples into separate streams, which in turn may result in incorrect deductions. For example, in our case, by splitting the main RDF stream with respect to the source of the monitored data, we deliberately broke existing links between the PostgreSQL service and its backup service PGBackups. Since respective triples are handled in isolation from each other, it is now impossible to detect situations when some of the established client connections are actually connections established during a temporary, short-lasting back-up process. Therefore, careful design of the fragmentation logic is crucial in order to confirm that no valuable data is misplaced or lost.

7.4 Experimental results

In this section, we will evaluate the presented approach and the described use case scenario with a number of experiments aiming at demonstrating the performance of the framework with respect to several configurable parameters. When running the experiments, we employ the following approach – we first run the default configuration, which is a configuration under which we expect the EXCLAIM framework to operate normally. Then, we will simulate an increasing workload by tuning certain configurable parameters to see how the framework performance changes. Table 7.2 summarises parameters, which may affect the performance of the framework. In our conducted experiments, we have used two configurable parameters – namely, the number of registered C-SPARQL queries and the triple generation rate (i.e., the number of triples generated per second).

7.4.1 Experimental setup

It is worth explaining that to achieve an intensive triple generation rate and feed the EXCLAIM framework with hundreds and thousands of triples per second we relied on simulation techniques. More specifically, to achieve a generation rate of 1,000 RDF triples per second, we would need several hundreds of Destinator instances accessed and used by users simultaneously. Unfortunately, Destinator is not at that advanced stage yet, so we had to simulate this workload. To do so, we first recorded 1,000,000 RDF triples, collected during the usual operation of a single Destinator instance running on Heroku, in a log file. Then, the simulator was configured to 'replay' these recorded values at a required rate – that is, we were able to stream monitored data at the rates of 1, 10, 100, 1,000, 5,000 and 10,000 triples per second respectively. Arguably, simulated experimental data might not completely and accurately reflect the actual behaviour of a simulated system in general; nevertheless, in this particular context of the presented case study, the general *shape* of the curve on the graphs is more important than absolute values, and therefore we assume the simulation error to be non-significant.

Table 7.2: Parameters affecting the performance of the framework.

Configurable parameter and its range	Description
Window size (from 1 ms)	The size of the window operator in the C-SPARQL queries. A larger window size typically(but not necessarily) results in more values to be queried. In certain applications, it may be important to keep all values arrived over the last hour or day to detect patterns. As demonstrated by our experiments, the 'wider' the window, the more RDF triples the C-SPARQL engine has to keep in memory and the slower it performs.
Window step (from 1 ms)	The step of the window indicates how often the window operator has to 'tumble'. If the window step is larger than the window size, then there will not be overlaps between to neighbour consecutive windows. This parameter does not directly affect the performance of the EXCLAIM framework.
Triple generation rate (from 1 to 10,000 triples/sec)	The rate, at which RDF triples are generated and sent to the monitoring messaging queue, from which they are then picked up the EXCLAIM framework. As demonstrated by our experiments, the higher the generation rate, the more loaded the framework gets (i.e., each window, defined in terms of time unit is more 'dense'), and the slower its performance becomes.
Number of registered C-SPARQL queries (from 1 query)	This parameter defines how many queries RDF triples on the stream have to be evaluated. The more registered C-SPARQL queries, the longer it takes to check data with respect to each of them and, accordingly, the slower the performance becomes.
CPU (2 configurations: Laptop Lenovo S540: Intel® Core™ i7-4500U Processor 2.00GHz-2.60GHz & Desktop Personal Computer (PC): Intel® Core™ i5-2310 2.90GHz-3.10GHz)	We have used two CPU configurations to achieve consistent, reproducable and more precise results. In general, a more powerful processor enables faster execution of the framework.
Memory (from 256 MB)	In the first instance, more memory allows to keep more RDF triples to support RDF stream processing, and avoid 'out of memory' exceptions.
Richness of the knowledge base (from 1 axiom)	This parameter affects the 'static' part of reasoning. In the first place, richness of the knowledge base means the number of axioms in the TBox.
Number of critical situations at a time (from 1 instance)	This parameter also concerns the 'static' part of reasoning. The number of critical situations detected at a time – that is, the number of triples detected by the C-SPARQL engine and sent to the OWL/SWRL reasoner – corresponds to the ABox of the ontology.

7.4.2 Conducting the experiments with the initial deployment

The default experimental setup was the following: we registered 21 C-SPARQL queries, defined the window step as 100 ms (i.e., it is evaluated 10 times per second), the window size as 1 minute. We run this setup on two machines (i.e., the PC and the laptop) with a goal to measure detection time – time between the moment when the last RDF triple, representing a complex critical pattern, was generated and the moment, when this critical situation was detected by the framework. We simulated the workload on the framework by increasing the triple generation rate. Figure 7.4 illustrates the curve of increasing detection time with respect to increasing workload.

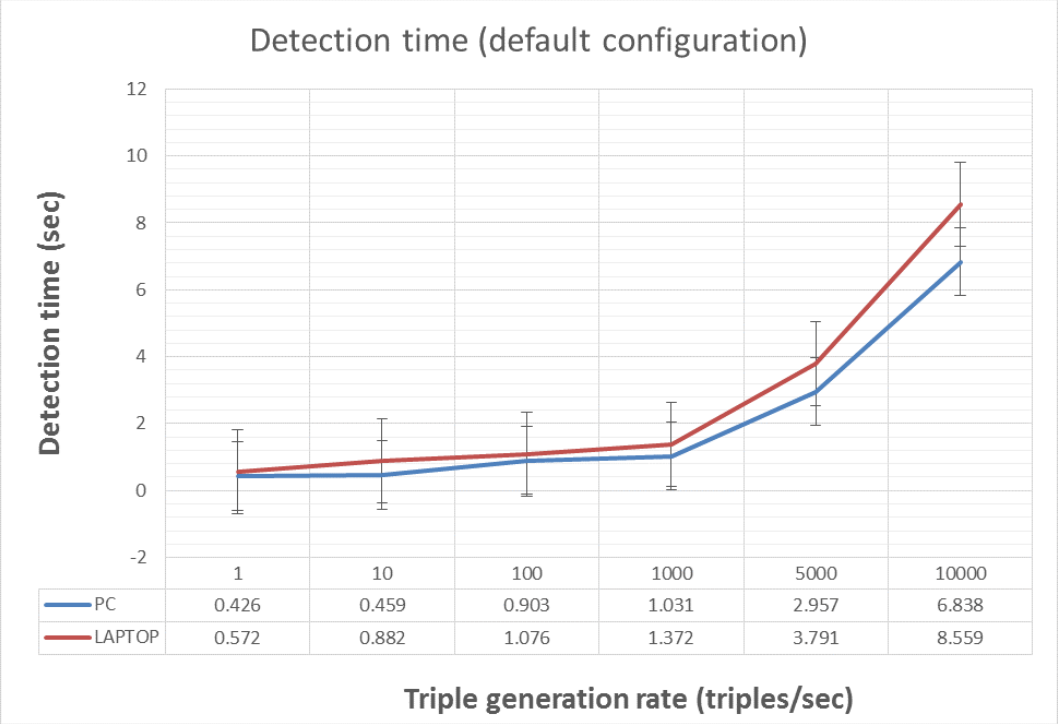

Figure 7.4: Detection times under the default configuration.

In the second part of the experiment, we increased the number of registered C-SPARQL queries. With the increased number of queries, we aimed at reflecting on the existing heterogeneity of monitored instances present on CAPs, each of which requires individual queries to be applied. Accordingly, the framework has to evaluate incoming values against a larger set of queries, which inevitably affects its performance. Figure 7.5 illustrates increased detection time with 100 registered C-SPARQL queries.

Detailed discussion of the obtained performance results is presented in Sec-

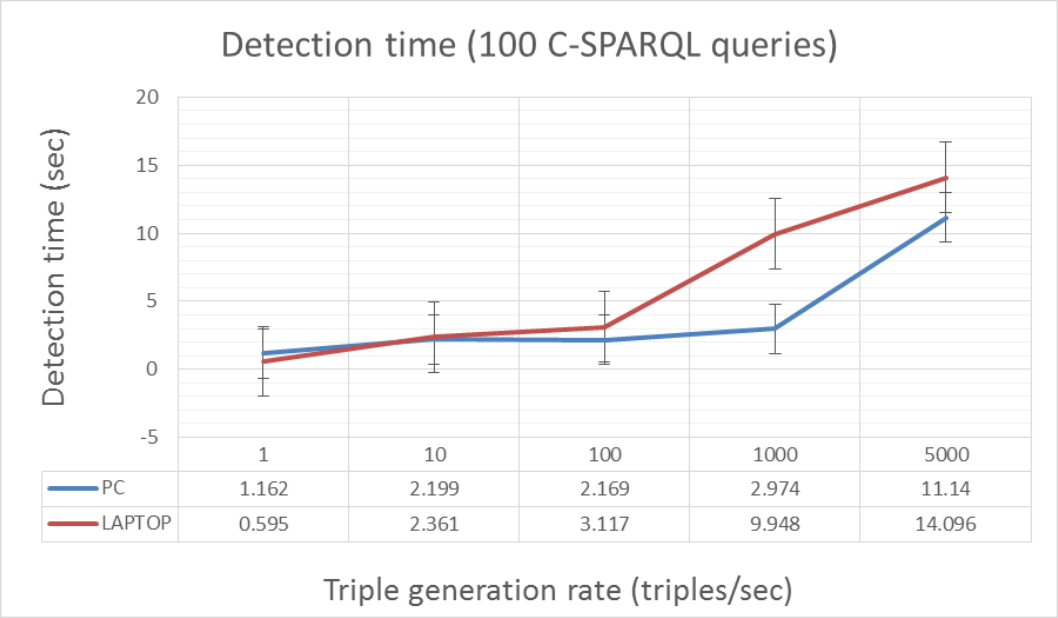

Figure 7.5: Detection times (100 registered C-SPARQL queries, window step 100 ms, window size 1 min).

tion 8.1. Nevertheless, we can conclude with the following observation – as the workload is increasing, the framework is facing considerable challenges, which stem from the volume, velocity and variety of collected data. Scalability of the framework is challenged by the emerging Big Data, and appropriate actions have to be taken to address this issue.

7.4.3 Conducting experiments with the IBM Streams deployment

In this section we provide the reader with experimental results obtained from running a parallel deployment of the EXCLAIM framework and splitting the main RDF stream into five separate sub-streams. The goal in these experiments, similarly to the experiments explained in the previous chapter, was to detect an increasing number of client connections to the database service. The only difference here was that we did not take into account a potential presence of the backup process. The reason for this was the requirement to process RDF streams coming from different services separately, and therefore it was impossible for triples from the PostgreSQL and PGBackups services to appear on the same sub-stream.

Then, similarly to the initial experiments, we increased the number of C-SPARQL queries registered with the framework. With an increase in the number of registered queries, the advantages of the parallelised deployment are clearer. The experimental results indicate improvements in the framework performance. The

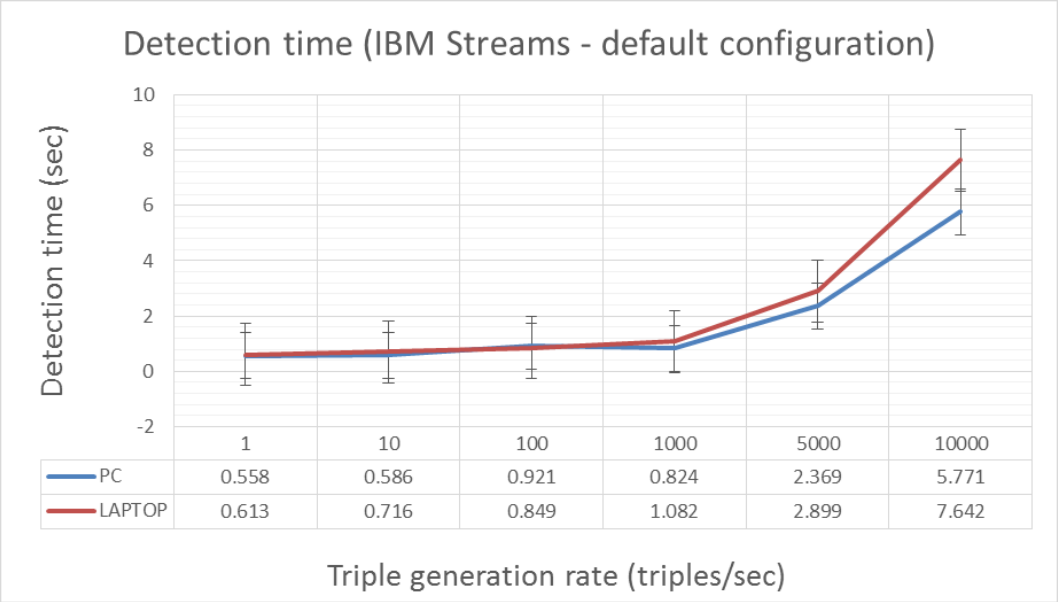

Figure 7.6: Detection times on IBM Streams (21 registered C-SPARQL queries).

graphs (see Figures 7.6 and 7.7) demonstrate a noticeable drop in detection time. The fragmentation logic on the IBM Streams deployment was rather simplistic. In practice, however, fragmentation algorithms may be somewhat more sophisticated and intelligent. Therefore, we may conclude with the following statement – IBM Streams has the potential to address the scalability issues faced by the framework through splitting data streams and applying parallel processing to resulting sub-streams.

7.5 Summary

In this chapter, we demonstrated how the EXCLAIM framework operates by presenting a case study. This case study focuses on the Destinator application, which was deployed on Heroku and connected with 5 add-on services offered by the platform marketplace. The main goal of the considered use case scenario was to demonstrate the analysis capabilities of the framework from perspectives of the two main stake-holders – i.e., the CAP provider and the CAP consumer. Additionally, using code snippets, the chapter demonstrated how the main components function, and how monitored data is first transformed and then flows within the framework. The performance of the framework was also evaluated with respect to configurable parameters – namely, the number of registered C-SPARQL queries and triple generation rate. By tuning these parameters and increasing the workload on the framework, we observed a considerable drop in the framework's per-

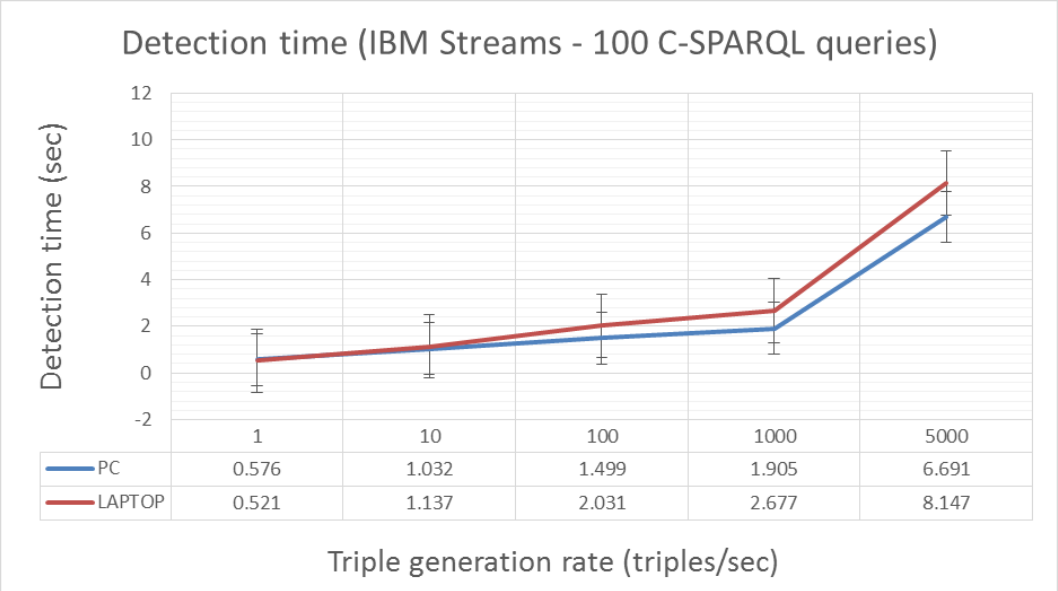

Figure 7.7: Detection times on IBM Streams (100 registered C-SPARQL queries).

formance. This drop, caused by the overwhelming amount of triples which have to be simultaneously processed, led us to investigation of Big Data processing techniques to be applied in this respect. The potential of the framework to scale was demonstrated by deploying it on top of the IBM Streams. This integration enabled us to parallelise the main RDF and evaluate the incoming triples by five separate instances of the EXCLAIM framework independently, thereby improving problem detection time. In the next chapter, we will evaluate and discuss the obtained experimental results, and compare the framework with existing approaches in terms of its performance, scalability, and extensibility.

Part III

Evaluation, discussion, and conclusion

The last part of this thesis summarises all the material, which has been described so far, in a more structured manner. Chapter 8 evaluates the approach against the current state of the art with respect to performance, scalability, and extensibility. The chapter also brings together explicitly the potential benefits and limitations of our proposed approach, and finally outlines several areas for further work. Chapter 9 concludes the whole thesis with an overall summary of the presented ideas. It evaluates the accomplishment of the issues raised in the introduction, summarises our main contributions, and outlines potential directions for further research.

Chapter 8

Evaluation and discussion

In this chapter we summarise the main aspects of the presented research in a more structured manner. First, we evaluate and discus our approach (i.e., performance, scalability, and extensibility) with respect to the state of the art. Next, we list potential benefits of the proposed approach so as to explain why each particular feature of the approach is of benefit – that is, how and what problem it solves. Then, we continue with potential limitations of the approach, which are listed and summarised in a similar manner.

8.1 Evaluating performance, scalability, and extensibility

Before evaluating and discussing our own approach, we first brief the reader on the current state of enabling data monitoring and self-management in service-based cloud environments – that is, what tools are offered by CAP providers and what actions consumers are required to take in this respect.

The topic of self-governance in service-based cloud platforms is relatively novel, and, as we have seen in the literature survey, neither CAP providers not third-party researchers offer targeted solutions. The current baseline is the following – CAP consumers (typically, software developers) are required to implement their own analysis and problem detection components themselves to suit requirements of their individual applications. They are offered a wide range of existing commercial (i.e., offered by the cloud platform itself or third parties) and open-source monitoring frameworks to collect data metrics at the infrastructure, service composition, and application performance levels. These frameworks are, however, disparate and heterogeneous, and need to be properly integrated together. Additionally, they only offer basic data aggregation tools, which cause software developers to implement a dedicated analysis engine from scratch. This is typically

achieved by hard-coding policies and enforcement mechanisms within the application source code, resulting in application-specific, non-extensible, non-scalable, and hardly reusable components.

The EXCLAIM framework has the potential to change this situation by introducing an approach, which demonstrate considerable performance results, scalability and extensibility. These three criteria are taken into consideration, when evaluating and discussing the potential of the EXCLAIM framework to address the challenge of problem detection in the context of CAPs with respect to the state of the art in the considered domain.

8.1.1 Performance of the EXCLAIM framework

It has to be noted that the challenge of evaluating and comparing performance of various approaches against each other can only be conducted using a common benchmark – that is, compared approaches have to run in the same conditions against the same data sets. This, however, is not feasible in the case of the EXCLAIM framework, which currently offers monitoring support, which is the only one of its kind to a certain extent.

Nevertheless, it is still possible to evaluate the performance of the framework based on the experimental results, presented in Chapter 7. From the diagrams (see Figures 7.4 and 7.5) it follows that the framework starts facing considerable difficulties when processing incoming RDF triples typically after the triple generation rate exceeds 100. On average, at the rate above 1,000 triples generated per second, detection time reaches values, which might be unacceptable as far as timely reaction in the context of mission- and time-critical applications is concerned.

To provide a deeper understanding of what computational workload the EXCLAIM framework has to handle, we now estimate the amount of data being processed by the framework. The size of a serialised RDF triple being sent over the network from the monitored entities in the form of a JSON string, ranges from 200 to 250 bytes. We take 225 bytes as an average value, which means we validated the framework against the rate of 2.15 MB of incoming data per second (under the most intensive setup – 10,000 triples per second). The C-SPARQL engine was configured to keep in memory the most recent window of values arrived in the last 60 seconds and initiate query evaluation every second. This in turn means that every single second situation assessment had to be performed against a data set of 128.75 MB. Moreover, with this speed, the amount of data passing through the framework per hour and per day equals to 7.54 GB and 181.05 GB respectively.

To compare, Google App Engine reports on the following activity in their data centre (Hellerstein, 2010, Di et al., 2013). A computation cluster consisting of

nearly 12,500 nodes is monitored to provide run-time statistics on the current state of the system. The sampling rate is 5 minutes, monitored metrics are CPU and memory utilisation, disk space, and disk time fraction, and each sample is stored in the Comma-Separated Value (CSV) format and occupies approximately 26 MB of disk space. Arguably, the sampling rate and the set of infrastructure-level metrics seem relatively easy to handle. At the platfrom level, however, the situation is more challenging. For example, Cronycle,[1] a Heroku-based content curation application, processes 30 GBs of images and videos, follows 127,000 twitter accounts looking for new content, and stores 240,000 articles on an average day. Applauze,[2] another Heroku-based app for event discovery and ticketing, reports 130,000 client sessions on a daily basis. Similarly, Forever Living[3] – a health and wellness products company – runs its e-commerce platform on Heroku to enable real-time operation in over 150 countries; it reports 86 million rows of stored data with 5 million rows of data processed daily. All these data-intensive activities are backed up with Heroku's add-on services, which in this context also act as Big Data generators in their own right, and whose run-time operation can potentially be monitored and analysed by treating individual services and applications as software sensors.

The presented statistical data is intended to demonstrate that the presented use case arguably can be compared to real-wrold cloud workload. Furthermore, it has to be noted that the case study was only focusing on a single add-on service of a single application – that is, in reality the volume and the pace of monitored data, generated by thousands of deployed applications coupled with numerous add-on services, may be considerably higher. Also, some scenarios may require window operators to keep a somewhat higher number of most recent values (e.g., arrived within the last ten minutes, not just one). Given the extreme heterogeneity and large number of entities to be potentially monitored, the number of corresponding C-SPARQL queries and SWRL rules will grow accordingly, and, as a consequence, the reaction time will increase. Arguably, these presented factors are sufficient to draw the following conclusion – scalability of the approach is challenged by the Big Data problem, associated with volume, velocity and variety of data generated within CAPs, and requires corrective actions to be taken.

[1] `https://www.heroku.com/customers/cronycle/`
[2] `https://www.heroku.com/customers/applauze/`
[3] `https://www.heroku.com/customers/foreverliving/`

8.1.2 Scalability of the EXCLAIM framework

Chapter 7 explained how the EXCLAIM framework can benefit from applying an existing Big Data processing solution. In particular, it demonstrated the potential of the IBM Streams platform to underpin a parallelised deployment of the framework, where the main RDF stream is fragmented into several sub-streams, which are then processed in parallel.

The parallel deployment on IBM Streams was then benchmarked against the same data sets as in the initial (i.e., non-parallel) deployment, thus enabling us to evaluate the two settings. Similar to the initial setting, the number of C-SPARQL queries registered with the framework increased, and performance dropped. This drop, however, is considerably less, as demonstrated by the diagrams in Figures 7.6 and 7.7. According to the obtained results, same critical situations are detected up to two times faster when running five parallel sub-streams, as opposed to the single-stream processing mode.

Arguably, the described increase in the framework performance is situation-specific, and may vary from one scenario to another. Potentially, there may be situations, where stream fragmentation will not bring noticeable results, or will even make things worse by applying unnecessary fragmentation and splitting semantically-connected triples into separate streams, thereby missing a potentially critical situation.

In the considered example, the fragmentation logic on the IBM Streams deployment was rather simplistic – i.e., a special splitting operator is responsible for routing incoming RDF triples to a dedicated sub-stream depending on based on which add-on service they are coming from. In practice, however, fragmentation algorithms may be somewhat more sophisticated and intelligent. Therefore, we may conclude that IBM Streams has the potential to address the scalability issues faced by the framework through splitting data streams and applying parallel processing to resulting sub-streams – this process, however, has to be taken with care and dedicated fragmentation logic has to be thoroughly designed and applied.

When comparing performance and scalability of the framework against existing approaches, there are two aspects, which need to be taken into consideration:

- The EXCLAIM framework addresses the challenge of service monitoring and problem detection in service-based cloud environments – a challenge, which has not been extensively addressed by the state-of-the-art approaches. So far, few approaches focus on the PaaS segment of cloud computing, and even fewer approaches focus on service-based compositions deployed on CAPs. In these circumstances, comparing the EXCLAIM approach with the

state of the art with respect to performance and scalability seems infeasible. Despite the fact that the framework doe snot scale 'out of the box', we demonstrated how improved scalability can be achieved by using the IBM Streams platform.

- It also needs to be noted that performance and scalability have not been seen as the primary goal of the presented EXCLAIM framework. Admittedly, there are existing and highly-optimised solutions (e.g., third-party solution Nagios and New Relic, or cloud-native solutions Amazon Cloud Watch and Stackdriver Monitoring), which can offer considerably faster mechanisms for data collection and aggregation. These existing approaches, however, do not offer solutions for data analysis, which remains the responsibility of the cloud consumer. One of the main benefits of the EXCLAIM framework is the automated analysis support, which enables timely detection of critical situations. The framework offers a built-in analysis component, responsible for problem detection – a feature, which is currently beyond the capabilities of the other existing approaches for data monitoring in the context of cloud platforms.

8.1.3 Extensibility of the EXCLAIM framework

Extensibility of the EXCLAIM framework refers to its ability to extend its functionality in a seamless and transparent manner so as to be used in new emerging circumstances. This assumes adding new (or extending existing) functionality with minimum interference with and influence on the usual operation of the framework. Extensibility of the EXCLAIM framework is underpinned by the two main features – namely, a declaratively defined knowledge base and modular architecture.

Declarative approach to defining the knowledge base

The declarative approach serves to separate the knowledge base (i.e., an architectural model, queries and policies) from the programming source code, responsible for querying potentially critical situations and enforcing respective detection policies. With such an approach, it is typically only necessary to modify the knowledge base, while avoiding changing and re-compiling the source code. In particular, in order to adjust the framework to a new emerging scenario, it is required to add corresponding CSO concepts, C-SPARQL queries, and SWRL rules, and reuse already existing 'triplification' connectors. As opposed to the state-of-the-art

approaches (e.g., cloud-native and third-party monitoring solutions), which typically rely on hard-coding analysis logic, the declarative approach employed by the EXCLAIM framework can be seen as a considerable benefit, which enables it to operate with minimum interruptions caused by potential modifications.

Modularity

In the context of the presented framework, the topic of modularuty is two-fold. First, the modularity principles have been applied when separating the main programming code from the knowledge base. Such a sepration of concerns noticeable simplifies modification of the knowledge base (even if it was not for the declarative definition). Nevertheless, separation of concerns is a common practice, and is typically achieved by all the relevant approaches employing declaratave domain-specific languages.

Second, modularisation has been applied to implementing the knowldge base *per se* as a set of linked elements. It makes it possible to extend the knowledge base (and, therefore, functionality of the framework) by third parties, who are responsible for a dedicated part of the detection policies. By applying the Linked Data principles and breaking down the knowledge base into multiple connected components, it is possible to delegate maintenance tasks to the respective responsible parties. As a result, third-party add-on providers, who are expected to be more familiar with potential critical situations concerning their offered services, can modify their respective linked extensions transparently to the CAP provider and consumers. Such a distributed modular structure has not been implemented or even discussed by existing approaches, and admittedly contributes to the overall extensibility of the EXCLAIM framework.

8.2 Potential benefits of the proposed approach

This section lists and summarises potential benefits demonstrated by the described EXCLAIM approach.

Separation of concerns and declarative definition of knowledge

One of the main motivating factors and goals of the research work described in this thesis was the creation of a mechanism, which would:

- separate the definition of knowledge concerning diagnosis and adaptation policies, an architectural model, queries, etc., from the actual enforcement of these policies; and

- allow the definition of the knowledge base in a declarative, loosely coupled manner.

The first requirement on its own can be simply addressed with traditional, component-based programming techniques. That is, one can capture knowledge at the level of the programming source code in a separate component (e.g., a class, library, etc.). The second requirement, however, calls for applying more sophisticated techniques for policy definition. The SSW technology stack and primarily RDF, OWL and SWRL successfully addressed this requirement, and provided us with the possibility to declaratively define the knowledge base, and, as a result, modify it if needed dynamically during run-time, without recompiling and restarting the whole application system. In other words, with ontologies and rules separated from the platform/application programming code, it is easier to make changes to adaptation policies 'on the fly' and to maintain the whole framework in a stable, operational state.

With employing the SSW stack as an instrument for defining the knowledge base, comes a positive side effect. As we saw above, the Semantic Web research targets at making information on the Web to be both human- and machine-readable, with languages which are characterised by an easy-to-understand syntax, as well as the visual editor Protege for effortless and straight-forward knowledge engineering. OWL ontologies are known to be used in a wide range of scientific domains (for example, see (Rubin et al., 2008) for an overview of biomedical ontologies), which are not necessarily closely connected to computer science, and allows even for non-professional programmers (i.e., domain specialists) to be involved in ontology engineering. Similar benefits are expected in the context of our own work – that is, development of diagnosis and adaptation policies (at least partially) can be undertaken by non-technical domain specialists.

In-memory stream processing

Another primary goal of the EXCLAIM framework was to enable timely, run-time processing of data and minimise the delay between the moment when data is generated and the moment when it is processed. As opposed to static approaches, which assume permanent storage of data on a hard drive and its further processing, we employed a streaming approach, whose fundamental characteristic is the ability to process data in memory, and thus avoid unnecessary and rather expensive hard disk access operations.

It is worth noting that the static approach, although associated with higher reaction times, might also prove to be useful in situations, where the amount of

historical data to be taken into account when performing analysis is considerably large. Existing technological constraints make it impossible (or at least impractical) to keep in memory data monitored over the previous several hours or even days. For such kind of scenarios, it is much more convenient to record all the data first and then perform post-mortem analysis over this single large dataset. In contrast, in situations where the analysis process only depends on the very recent observations, captured within last seconds and minutes, the in-memory stream processing is a key to success.

As we demonstrated with the case study, we managed to achieve timely execution of the framework, such that the time required to detect and react to a critical situation is measured in seconds – a time frame, which, we believe, is affordable so as to plan and apply all necessary reactive actions.

Increased opportunities for reuse

One of the main characteristics of ontologies is their public openness and extensibility. Once implemented and published on the Web, ontologies can be imported and immediately re-used or extended with necessary domain-specific concepts by any third party. This reduces time and effort to develop an ontology from scratch, and exempts ontology engineers from 'reinventing the wheel'.

Publishing the CSO and experimental datasets on the Web, and linking them with other existing ontologies (i.e., primarily with the SSN ontology) may contribute to the development of the Linked Open Data (LOD) – a global giant data set published on the Web, in which individual elements are linked with each other by means of URIs. We further discuss this in Section 9.2.

Increased automation and reliability

Policy enforcement mechanisms already exist in the form of automated reasoners, and the EXCLAIM framework aims to build on these capabilities. By employing the SSW stack, we relied on the built-in reasoning capabilities of OWL and SWRL. Since the reasoning process is automated and performed by a reasoning engine (e.g., Pellet), it is expected to be free from so-called 'human factors' and more reliable (assuming the correctness and validity of ontologies and rules).

To develop the EXCLAIM framework, we used an existing Java implementation of the Pellet reasoner, which required from us adding just several lines of code, responsible for initialisation of the library and the CSO, and actual invocation of the reasoner returning instances of detected critical situations (if any). The whole analysis routine was solely handled by the reasoner based on the CSO and SWRL

rules in a transparent manner. Conversely, consider a situation when the diagnosis and adaptation logic is hard-coded in the programming code with numerous `if-then` and `switch-case` operators. As the knowledge base grows in size and complexity, accurate and prompt maintenance of such a tangled definition becomes a pressing concern. Arguably, a complicated and extensive knowledge base, defined with OWL ontologies and SWRL rules, may also be associated with certain difficulties so as to how to maintain it.

Extensible architecture

Existing sensor networks typically comprise a vast number of sensing devices spread over a large area (e.g., traffic sensors distributed across a city-wide road network) and have the capacity to be easily extended (as modern cities continue to grow in size, more and more sensors are being deployed to support their associated traffic surveillance needs). Once new sensors are connected to the existing network, they are ready to report on the current situation. A similar extensible architecture was introduced with our EXCLAIM framework. Thanks to the decoupled architecture for collecting monitored data, new software sensors can be seamlessly integrated into the framework and start sending sensory observations to the main monitoring channel. With the help of declarative definitions, the knowledge base can also be easily extended to cover newly-added components in a transparent and non-blocking manner. The same applies to the reverse process – once old services are retired and do not need to be monitored and analysed anymore, the corresponding policies can be seamlessly removed from the knowledge base so as not to overload the reasoning processes.

More specifically, the extensible architecture paves the way for creating a comprehensive EXCLAIM framework, which would incorporate all add-on services and applications of the cloud ecosystem. Throughout this document we have specifically focused on monitoring and analysis at the PaaS segment and in CAPs. However, it can be potentially extended and applied to the IaaS and SaaS levels as well. That is, it is possible to monitor CPU/memory utilisation and network bandwidth utilisation by equipping the underlying infrastructure with appropriate software sensors and integrating them into the EXCLAIM framework. Similarly, embedding application-specific sensors at the software level can help perform monitoring and analysis activities in terms of business goals.

Non-intrusive monitoring

A considerable advantage of the proposed EXCLAIM framework is the non-intrusive approach to data collection. The framework relies on already existing APIs provided by add-on services to collect metrics. Typically, there are also already-existing client libraries, which enable interaction with an add-in service and may also serve to extract relevant data.

As demonstrated by the Destinator case study, we relied on service APIs to collect raw data from the application and add-on services, which did not involve explicit changes to the source code. Based on the assumption that the EXCLAIM framework is an integral part of the CAP and acts as a trusted entity, we assumed to posses user credentials to collect monitored data emitted by the a particular instance of an add-on service associated with the current user. In other words, the EXCLAIM framework is assumed to be equipped with sufficient access rights to platform components, services and applications.

'Non-intrusiveness', however, does not apply to the requirement to implement service connectors – software components responsible for translating raw data formats into the semantically-enriched RDF representation. These connectors need to be implemented from scratch (at least in the beginning, and can be re-used afterwards).

Existing solutions and best practices

Treating a CAP as a sensor network allowed us to re-use existing solutions, developed and validated by the Semantic Sensor Web community, in the context of data monitoring and analysis of streaming heterogeneous sensor data. This enabled to perform situation assessment by providing meaning for sensor observations (Sheth et al., 2008), by expressing sensor values in the form of RDF triples, thus providing more meaningful descriptions and enhanced access to sensory data. Moreover, this extra layer helped to bridge "the gap between the primarily syntactic XML-based meta-data standards of the SWE and the RDF/OWL-based meta-data standards of the Semantic Web" (Sheth et al., 2008). The advantage of this approach is that semantically-enriched sensor data helps to homogenise various data representation formats existing in the SWE and also facilitates more intelligent analysis of observed sensor values by applying formal reasoning.

The interpretation of CAPs as sensor networks, however, does not restrict one from using other existing solutions besides the SSW techniques. For example, for modelling purposes, one might employ the more light-weight SensorML vocabulary to model a software sensor network without support for semantic inter-

operability. Given that semantic annotations and the RDF format may result in undesired overheads in terms of network transportation, using SensorML might be more appropriate as far as network latency is concerned.

Conceptual architecture and platform independence

The conceptual architecture of the EXCLAIM framework (and the even more abstract interpretation of CAPs as sensor networks) can act as a high-level conceptual model for creating monitoring frameworks, which is independent of the underlying technology and implementation. In other words, the model does not constrain how sensors are designed and implemented, and how communications between sensors and the monitoring components takes place – depending on the context and technical requirements, it may be performed via a publish-subscribe mechanism, an enterprise messaging bus, via the Simple Object Access Protocol (SOAP) and Representational State Transfer (REST) protocols, etc. Our framework also does not constrain the underlying platform and programming languages – Java, .Net, Ruby, etc., are all equally acceptable.

Application profiling

Having introduced the notion of service and application criticalities, we enabled a more fine-grained and flexible approach to treat CAP entities during the monitoring and analysis process. Service criticalities define the overall criticality of an application deployed on the CAP and coupled with the corresponding services. We refer to this resulting overall application criticality as an application profile, which determines a set of diagnosis and adaptation policies to be applied in the given analysis context.

With application profiling, it is possible to modify criticalities of individual services if needed. For example, it is possible to classify an application as a data-intensive one, which therefore calls for special attention and treatment to respective data storage service it is connected with. It means, that the thresholds, which actually represent danger levels (e.g., critical number of simultaneous client connections), have to be lowered so as to detect a potentially critical situation as early as possible, and possibly take certain preventive actions.

Self-containment and modularity

In the research work presented in this thesis, we were motivated to enable declarative and loosely-coupled mechanism for defining and modifying diagnosis and adaptation policies. We aimed at separating definition of policies from their actual

enforcement. For this reason, we followed the SSW approach, which enabled us to define the knowledge base in a declarative manner by means of OWL ontologies and SWRL rules. However, as the resulting knowledge base is growing in size and complexity, it may result in a considerably heavy-weight ontology, and therefore can slow down the reasoning process.

To address this challenge, we applied principles of modularity, which are known to be effective in overcoming complexity in modern information systems (Baldwin and Clark, 2000). By breaking down the whole knowledge base into several parts, we enabled the reasoning mechanism to pick only the necessary elements of the knowledge base so as to avoid keeping the whole heavy-weight ontology in the memory when performing reasoning activities. To some extent, a similar principle is implemented in various online maps and navigator software – at a given moment users are typically expected to work with a specific sub-set of available maps (e.g., a map of city or a country), rather than with the whole collection. Accordingly, in order to increase performance and network bandwidth, software usually load and keep in memory only this specific sub-set.

It is worth noting that at its extreme, the ontological knowledge base can be potentially broken into hundreds of linked parts, where each sub-set holds only a single logical statement. In this case, potential benefits are outweighed by the requirement to 'crawl' all the references and assemble all these distributed broken chunks into a single ontology. Therefore, partitioning has to be balanced, and this process has to be handled with care.

Another reason for employing the modular architecture was the necessity to enable third-party service providers with capabilities to modify diagnosis and adaptation policies. Since CAP administrators may not be in a position to define the diagnosis and adaptation logic, which concerns third-party services registered with the CAP marketplace, this responsibility has to be shifted to service providers. From the CAP provider's point of view, third-party services are seen self- and context-aware 'black boxes'. From this perspective, they become self-contained entities, which are equipped with sensing capabilities and self-reflective knowledge – that is, services themselves are aware of the potential critical situations they may face, as well as the diagnosis and adaptation rules to be applied in this respect. As a result, CAP providers are exempted from additional responsibility of maintaining the policy base – a job, at which they are not necessarily competent – and, on the other hand, these policies, defined by respective service providers, are more accurate and prompt. Moreover, further elaborating on the benefit of having an extensible architecture, with the knowledge base, defined in a modular fashion, the process of adding/removing software sensors from the

sensor network becomes even more transparent and less interruptive.

In this context, we have utilised Linked Data principles to create a two-tier ontological architecture, consisting of the core ontology (i.e., the CSO) and multiple linked extensions to it. The primary goal of Linked Data is to enable discovery and sharing of semantically-enriched data over the Web using standardised technologies, such as URIs and RDF (Bizer et al., 2009). This enabled us to separate various pieces of adaptation policies between CAP owners and third-party service providers.

The main benefits of Linked (Open) Data are that it is sharable, extensible, and easily re-usable. In the context of a distributed ontological framework for self-governance policies, we can also distill the following additional benefits:

- Linked extensions are distributed and easily accessible over the Web by means of URIs and/or SPARQL endpoints. In this sense, software services become 'self-contained' as they inform the EXCLAIM framework about their diagnosis policies by providing a link to the corresponding set of rules. The EXCLAIM framework does not need be aware of them in advance, but can access them at run-time using Linked Data principles and 'crawling' provided URIs.

- Linked extensions are easily modifiable. Since third-party service providers have full control over their segment of policies, they can seamlessly adjust them so as to reflect ongoing changes.

- Linked extensions are potentially re-usable across multiple CAPs. Indeed, it is quite common for third-party service providers to offer their services on several CAPs. For example, the messaging queue service CloudAMQP[1] is offered on 10 different CAPs (including AWS, Heroku, Google Cloud Platform, etc.). Accordingly, under certain assumptions one and the same policy definition can be re-used across all of those CAPs.

8.3 Potential limitations of the proposed approach

Having listed the main benefits of the present research work, we also have to summarise associated limitations, as our approach is not a 'silver bullet', and still may suffer from certain issues.

[1] http://www.cloudamqp.com/

Need for unified data representation

The domain of data monitoring and analysis within CAPs is characterised by an extreme heterogeneity of the data sources generating data. This data may come from various physical (e.g., data centres and servers) and logical locations (e.g., log files, databases, SQL queries). This data may also come in non-structured (e.g., text file), semi-structured (e.g., data expressed in JSON or XML formats) or structured forms (e.g., relational databases). There also exists what we call a semantic heterogeneity – that is, difference in data representation at the semantic level, when various applications, for example, use different terms and names when storing data in the same database (i.e., in the same structured format).

All these call for a unified way of representing data so as to enable its processing in one central location, based on a coherent knowledge base. The same challenges exist in the domain of the Sensor Web (that is why the whole SWE initiative was launched), and effective solutions came from the domain of the Semantic Web, which offered rich and expressive ontological vocabularies to describe sensors and model sensory data, and the RDF format for unified data representation. Accordingly, in our own problem domain we applied the same technique – heterogeneous sensory values, generated by various software sensors, were expressed using the CSO vocabulary in the form of RDF triples and streamed to one central processing component.

A negative side effect of employing such an additional meta-description language for heterogeneous values is the increased computational overhead, associated with excessive data serialisation. Every time a serialised string object, representing an RDF triple, is sent over the network, it has to be first marshalled into textual form, and then un-marshalled back into the RDF object. Besides that, values converted into RDF representation (on average 225 bytes per RDF triple in a serialised JSON format) may occupy more space compared with the source format.

To a certain extent, this shortcoming stems from the problem domain itself, rather than our approach. The existing heterogeneity requires employing an additional uniform representation data format. Unless we wanted to create individual monitoring and analysis components for every single type and format of data present within the CAP, we had no other option than to introduce a single, unified vocabulary – an OWL ontology.

RDF stream processing is not standardised yet

As we explained it in Section 3.2.3, the RDF stream processing research is still in its infancy and only making its first steps towards being a fully-standardised part of the Semantic Web technology stack. Currently, there exist several disjoint approaches to processing streams of RDF triples, which may share the same general idea, but differ in operational syntax and semantics of the query languages they are using. As a result, existing RDF stream processing implementations, proprietary and often platform-dependent, are hardly compatible with each other. For example, while experimenting with the prototype version of the EXCLAIM framework, we attempted to employ another RDF stream reasoning engine to see how easy it can be plugged in to replace the C-SPARQL implementation. Our experiments showed that this process is not straightforward and requires considerable efforts to: i) refactor the programming code; and ii) rewrite the query set itself to align it with the new syntax.

Nevertheless, the standardisation process was launched with the support of W3C RDF Stream Processing Community Group, which was established and is being run by the main active contributors in the respective research area. We can reasonably hope for truly standardised RDF stream data models and query languages in the near future.

Immature reasoning support in RDF stream processing

This shortcoming stems from the previous one, and is a result of the immature state of the RDF stream processing research in general. It is worth noting that formal reasoning is not a vital, fundamental feature of an RDF steam processing engine, whose primary function is to extract individual triples and complex patterns from an RDF stream. However, the whole research area and the term Stream Reasoning coined by Della Valle et al. (2009) were initially expected to provide support for formal reasoning to the same extent as the traditional static approach does it. In particular, with SPARQL it is possible to query over inferred, implicit knowledge – for example, it is possible to resolve subclass dependencies when querying. These optimistic plans, however, have not come true yet (this is why the whole research area should be called RDF stream processing, rather than Stream Reasoning), and to date, existing RDF stream processing engines only support querying over explicitly defined RDF triples on a stream (albeit the W3C RDF Stream Processing Community Group is actively working in this direction).

Eventually, stream reasoning engines are expected to reach a level of maturity, where all reasoning tasks associated with problem detection in the context of the

EXCLAIM framework will be performed at the streaming step. Thereby, the static reasoning component will not be required at all. However, at the moment the existing expressivity of RDF stream processing is not enough to adequately represent diagnosis and adaptation policies with continuous SPARQL (i.e., C-SPARQL in our case) queries, and we had to extend the framework with the traditional, static reasoning mechanism based on OWL ontologies and SWRL rules. This, in turn, led to the next limitation associated with our approach.

Performance issues associated with formal reasoning

Stream reasoning research is still in its infancy, and the performance of dynamic reasoning over a data stream is an issue. Expressivity of a query language is known to be inversely related to its performance (Lanzanasto et al., 2012) – the more expressive queries are, the longer it takes to execute them. This affects both the scalability of stream reasoning systems, and the actuality and accuracy of the results obtained.

At the moment, the research community is putting its main efforts into integrating existing, disjoint approaches and developing a uniform and standardised approach to RDF stream processing. There seem to be individual successful achievements in this direction – for example, the CQELS continuous query engine demonstrates an advantage in query execution and performance, compared to C-SPARQL and EP-SPARQL (Le-Phuoc et al., 2011). These individual benefits, however, will need to be revisited, once a uniform, integrated approach is developed. The community will have to address a number of issues in this respect, including advanced reasoning support and performance issues, such as query optimisation, caching, and parallel query execution, etc.

To a lesser extent, this limitation also applies to the traditional, static reasoning over OWL ontologies. Formal reasoning based on DLs is known to be non-linearly scalable (Urbani, 2010). It means that in the presence of multiple SWRL rules, as well as the densely populated TBox and ABox of the CSO, the reasoning process will start facing performance problems. In this light, the role of the C-SPARQL engine is crucial – it performs initial filtering on incoming sensory data, so that the OWL/SWRL reasoner is not overloaded with an overwhelming amount of RDF instances.

Requirement to design a complicated algorithm for splitting the main stream into sub-streams

As opposed to simple stream processing approaches, which aim at detecting individual tuples on a data stream, CEP approaches, including RDF stream processing, rely on detecting patterns of events – that is, a combination of tuples, which have to be extracted and considered together, as a whole. Moreover, in order for continuous queries to trigger and extract a pattern of events, it is often required that individual evets not just appear on a stream within the specified time window, but appear in a certain chronological order as well. We refer to such patterns, which also include the chronological dimension of individual events, as sequences.

In this light, splitting the main RDF stream into several sub-streams is not a straightforward task – it may lead to situations in which various tuples, representing a complex pattern when taken together, may be separated from each other. As a result, respective queries will never trigger, and will fail to detect a critical situation. Therefore, the splitting algorithm has to be designed and implemented with care.

For example, in the use case scenario deployed on IBM Streams, as we described it in Chapter 7, we implemented the fragmentation logic in a rather simplistic manner. RDF triples were routed to a corresponding instance of EXCLAIM framework with respect to a Heroku add-on service, which generated them. Accordingly, five instances of the EXCLAIM framework were independently processing RDF triples from PostgreSQL, PGBackups, Logentries, IronWorker, and Memcached services. Even though the experimental evaluation demonstrated an increase in performance – that is, it took less time for the framework to detect a critical situation – we were unable to handle situations, where observations from more than one add-on service would help us identify a potentially dangerous pattern. In particular, by isolating data emitted by the PostgeSQL and PGBackups add-ons, we were unable to understand whether a critical number of simultaneous client connections to the database was indeed caused by an increase in number of clients trying to access it, or it was simply a short-lasting backup process establishing two connections during the backup procedure. It is obvious that a more sophisticated splitting algorithm is needed so as to address this kind of situations.

Portability issues of the prototype

A major challenge posed by the ideas presented in this research work is that implementing a monitoring and analysis mechanism based on this high-level abstraction is not straightforward. It is not possible to provide truly generic guidelines

as to how to implement the monitoring functionality, because implementation depends on the characteristics of a particular CAP (its architecture, supported programming languages, frameworks, execution environments, etc.). For example, our own work in this direction (described in (Dautov et al., 2013)) suggested that an early prototype of the EXCLAIM framework, implemented in Java Spring and deployed on VMWare's Cloud Foundry,[1] was portable to another cloud platform with considerable amount of time and efforts, due to its tight dependence on Cloud Foundry's built-in message queueing service RabbitMQ[2] as a means of transporting monitored values within the EXCLAIM framework.

8.4 Summary

This chapter discussed the presented EXCLAIM approach and compared it against the state of the art in the domain of data monitoring and analysing of service-based cloud systems with respect to the main three criteria – namely, performance, scalability, and extensibility. The chapter also summarised and discussed its potential benefits and shortcomings in a structured manner.

[1] http://www.cloudfoundry.com/
[2] http://www.rabbitmq.com/

Chapter 9

Conclusion

In the presented research work we raised the issue of the cloud platform governance, which is becoming of utmost importance, as cloud platforms compete, striving to deliver even wider selection of services and accommodate even more user applications. It is our belief that by offering rich, but increasingly complex tools for software development, cloud platform providers also ought to offer proper management and problem detection support for these tools. Similar to the IaaS level, where infrastructure resources are elastically provisioned and system outages are transparently handled, the PaaS level also has to include such self-management capabilities as part of their offering. In the first instance, such self-governance capabilities are expected to enable platform providers with more control over their constantly growing software ecosystems to support platform stability and optimal resource consumption. Furthermore, these capabilities are intended to exempt CAP customers from implementing this functionality themselves, and provide more visibility into how platform-hosted SBAs behave and perform. In this light, in the introductory chapter of this thesis we raised several research questions we aimed to address.

We answered these research questions with our proposed EXCLAIM framework for service-based cloud platforms. The main idea of the proposed approach is to encode monitored heterogeneous data using Semantic Web languages, which then enables to integrate these semantically-enriched observation streams with static ontological knowledge and to apply intelligent reasoning. The EXCLAIM framework follows the established MAPE-K reference model for self-adaptations, and a fundamental underpinning of this approach is the interpretation of service-based cloud platforms as distributed networks of 'software sensors' – that is, services, deployed applications, platform components, etc., which continually emit raw heterogeneous data to be monitored and analysed to support run-time situ-

ation assessment. This enabled us to apply existing solutions developed by the SSW community, which combines ideas from two research areas, the Semantic Web and the Sensor Web. This novel combination facilitates situation awareness through providing enhanced meaning for sensor observations. In particular, we were inspired by the SSN approach to express heterogeneous sensor values in terms of RDF triples using a common ontological vocabulary, and have created our own CSO to act as the core element of the EXCLAIM framework. The CSO is used to support both self- and context-awareness of the managed elements by describing the governance-relevant aspects of the managed cloud environment.

By addressing the research questions, our approach contributes to the domain of cloud self-governance, and in particular to the area of data monitoring and analysis on service-based cloud platforms. Main contributions include the presented approach itself, which allows monitoring and analysing heterogeneous data in an extensible, declarative, modular and scalable manner, the conceptual design of the EXCLAIM framework, the CSO, and novel concepts of service self-containment, service criticality, and application profiling. Our research also contributes to the state of the art in Software Engineering by demonstrating how existing techniques from several fields (i.e., Autonomic Computing, SOC, Stream Processing, SSW, and Big Data) can be combined in a novel way to create an extensible, scalable, modular, and declaratively defined monitoring and analysis solution.

9.1 Discussing contributions

We now summarise the main research contributions associated with the presented EXCLAIM approach. These are our findings, which have not been previously explored and discussed by the researchers, and can be potentially re-used by the wider research community should there be such a need.

1. **Novel concept of software self-containment**: this concept relies on interpreting software elements as logical sensors, and cloud platforms as distributed networks of such sensors, and allows for individual software sensors to be equipped with respective self-governance knowledge (e.g., self-diagnosis and self-adaptation policies) to enable decoupled, modular and distributed organisation of the knowledge base. Based on this interpretation, it becomes possible to design and implement monitoring frameworks in such a way that individual monitored objects act as independent, self-aware entities, which can be seamlessly integrated into an existing network of similar objects. As opposed to computationally expensive reasoning over a

potentially heavy-weight, 'monolithic' knowledge base, with such organisation, it is possible to limit analysis activities to a specific set of policies which only concern a specific scenario at hand, and thus minimise the amount of unnecessary computations and data transfers.

2. **The overall approach to data monitoring and analysis to support self-governance in service-based cloud platforms**: in the first instance, this contribution relies on through preliminary work, carried out before the EXCLAIM framework could be developed. This work refers to examining and analysing the problem domain of service-based cloud platforms – something, which has not been previously done by the research community, as there is little evidence of efforts in enabling any kind of autonomic behaviour at the level of CAPs. In the context of the presented work, we classified existing problems from several perspectives (i.e., the CAP provider, the CAP consumer, and the third-party service provider), their roles in the governance process, and identified challenges associated with each of these perspectives. Having classified the problem domain, we also distilled a list of functional requirements, which in turn helped to identify exiting techniques and solutions which could be potentially re-used in the context of our work so as not to 're-invent the wheel'. As a result, based on the existing similarities between problem domains, we were able to apply the SSW techniques. The proposed approach demonstrates capabilities for extensibility and scalability, as well as acceptable performance results. It can be seen as a combination of several technologies and research domains – namely, Cloud Computing, SSW, Stream Processing, Big Data processing, and Software Engineering – and potentially can be re-used and applied to other domains, where problems and challenges are similar to the ones identified in the context of CAPs.

3. **Conceptual design of the EXCLAIM framework**: apart from the general approach to data monitoring and analysis in CAPs, the conceptual design of the proposed EXCLAIM framework is seen as a research contribution, as it may serve as a reference model to implement similar frameworks using other components and technologies. That is, it acts a high-level generic architecture, where individual components can be implemented using various technologies. For example, as explained, there exist several competing technologies for RDF stream processing, each of which has its own advantages and therefore can be used to implement the streaming part of the framework. The conceptual design of the framework follows a modular architecture, which makes it possible to re-implement individual components in a

seamless manner with minimum interference to other components. Moreover, it is even possible to take the current version of the EXCLAIM framework and adjust it to another problem domain by replacing the knowledge base (i.e., the core ontology, C-SPARQL queries, SWRL rules) and software connectors.

4. **Two-tier Cloud Sensor Ontology**: the CSO acts as the core element of the underlying knowledge base, used throughout the whole process of data monitoring and analysis within the EXCLAIM framework. the upper tier of this ontology models cloud platform environments in a high-level generic manner – i.e., concepts belonging to this tier are expected to be applicable across a wide range of CAPs. This claim is supported by our own experience of engineering the ontology – multiple CAPs have been examined so as to identify commonalities among them, which constitute the upper tier of the ontology. As a result, the upper ontology can be extended appropriately to model a particular cloud platform (as it might be required by other researchers). In these circumstances, the lower tier of the ontology, which models the Heroku ecosystem, can be seen as an example of how the upper tier is supposed to be extended and used. Additionally, the lower tier can be potentially re-used as it is right now (or with minimum additions), should there be a suitable situation.

5. **Novel concepts of service criticality and application profiling**: these concepts serve to support a fine-grained, differentiated, and user-customised approach to data collection and monitoring. As there are three main roles involved in the process of platform governance, it was important to consider all three perspectives when designing the EXCLAIM framework, such that the involved parties (i.e., the CAP provider, the CAP consumer, and the third-party add-on provider) are given an opportunity to participate in the process of cloud platform governance. As a result, we introduced the concepts of service criticality and application profiling, which can be seen as a communication channel between CAP consumers, CAP providers, and third-party add-on providers. Consumers are not allowed to interfere with the back end of the EXCLAIM framework, including the knowledge base, but are essentially the ones who know the best how critical their applications are. That is, they are aware of the internal structure and business logic of their applications, and therefore might want to make the platform aware of what aspects of their software systems need to be cared after more thoroughly. Accordingly, by submitting an application profile, they let the

platform know of how critical their software is. To each of the criticality level there exists a corresponding set of policies, submitted to the CAP provider by third-party service providers, who are more familiar and aware of potential danger levels of their respective services. As a result, the cloud platform is now able to treat deployed software and connected add-on services more carefully if needed, based on the criticality profile submitted by the consumers and associated linked sets of policies submitted by the service providers. Such a differentiated, flexible and fine-grained approach to monitoring and analysing individual software elements within cloud platforms also facilitates more optimised utilisation of available resources by triggering analysis activities only when they are really required – that is, if an application is not profiled as critical, there is no urgent requirement to monitor its activity.

9.2 Further work

In this section, we outline several directions for future research. To avoid trivial and negligible (mainly implementation-related) details, we include only the most significant directions for further work.

Reducing redundancy in RDF streams and optimising sampling rates

As explained in Chapter 6, there is a considerable amount of redundant RDF triples, which are sent over the network to the EXCLAIM framework. More specifically, every sensor reading consisting of several RDF triples includes only one truly meaningful triple, which represents an actual quantitative value, whereas the rest only provide the semantic context of that value. That is, they describe what kind of value is that, to what service they belong, etc. These triples are fairly static – i.e., do not change as frequently as the 'meaningful' triple – and repeatedly sending same triples results in unnecessary network overheads.

A potential solution to address this issue is to distinguish between truly dynamic, 'meaningful' triples and static triples, describing the context of the measurements. The latter, therefore, can be added to the static background data set of the continuous query engine (e.g., C-SPARQL supports this feature) or the SWRL reasoner. To do so, it is possible to assign a dedicated stream, which would send and update the background data set with these fairly static triples at a much slower rate (e.g., once in an hour), thus considerably minimising the amount of triples sent to the main operational streams, as the ratio between 'meaningful' and

static triples in a single reading might get significantly low, as demonstrated by the examples in 6.2.

In connection with this, it is also important to consider the optimal rate for sampling and sending data to the EXCLAIM framework so as to avoid network overheads. There are several factors which need to be taken into account:

- RDF generation rate has to be lower than (or equal to) the data sampling frequency of the underlying software sensors. Otherwise, duplicate redundant values will be streamed for monitoring. For example, Heroku Postgres updates its statistical information up to 2 times per second, and the RDF generation rate has to be adjusted accordingly.

- It is also important to consider the behaviour of add-on services – that is, how dynamically they update their state. For example, database records are expected to be updated frequently, and therefore associated metrics (i.e., occupied disk space and number of connections) are supposed to be polled at a corresponding rate. On the other, a backup service is a one-off process, which might be executed on an hourly, daily or weekly basis – in these circumstances, polling the backup service every second whether it is currently running or not hardly makes sense.

- The third factor is the application profile, submitted by the CAP provider. Software may be classified in terms of its add-on service criticalities, which defines how sensitively the EXCLAIM framework has to treat particular applications and associated add-on services. Accordingly, this can also be taken into account when deciding on a specific sampling and RDF generation rate – more critical services require more frequent and up-to-date monitoring results, and vice versa.

These factors suggest that data sampling and RDF generation may vary and are specific to individual services. As a first step towards addressing this issue, we suggest surveying target services and identifying individual rates. Then, it will be possible to cluster them and set default rates for all services belonging to a particular cluster.

Semantic stream fragmentation

Detecting complex patterns of events is expected to be performed over a single stream. The situation changes when, for whatever reason, data is spread across several sub-streams, so that each of them provides a partial view on the overall situation. In the context of the research work, as we described in Chapter 7,

we fragmented the main RDF data stream into five sub-streams to achieve higher performance on individual sub-streams. However, analysing such partial observations may lead to flawed and inaccurate interpretations, and therefore has to be handled with care.

Accordingly, a potential direction for further research is the investigation of splitting and fragmentation algorithms for data streams. It is important to understand what the (semantic) links between individual tuples are, and whether these links can be 'broken' safely. In this light, applying principles of the semantic distance (Cross, 2004) may be of potential benefit. Briefly, based on the semantic distance (also known as *semantic relatedness*), we may decide whether two individual tuples have to be routed to the same sub-stream, or can tolerate being split into separate sub-streams.

Integration with existing planning and execution mechanisms

In Chapter 2.2, we introduced to the reader the notion of the MAPE-K loop for implementing self-adaptive systems as an underlying model for implementing our own EXCLAIM framework. In our work, however, we only focussed on the left-hand side of the loop – that is, on the monitoring and analysis components. To fully achieve autonomic functionality, it is necessary that the framework is integrated with convenient planning and execution components.

In this respect, a possibility of applying approaches, developed by the author's fellow researchers at SEERC, can be investigated. In particular, the EXCLAIM framework can be extended with the work of Bratanis (2012). Its main focus is on creating a self-management mechanism for service-based environments at multiple levels – i.e., hardware infrastructure, service composition and business process management. Such a cross-level mechanism would avoid managing service compositions at different levels in isolation, but rather would aim at more holistic, comprehensive management of service compositions. For example, when identifying a lack of hardware resources (i.e., at the infrastructure level), the planning mechanism would consider possible adaptation solutions to be applied at all three levels, not just the bottom one. Accordingly, analysis results obtained by the EXCLAIM framework would provide a diagnosis, based on which further cross-level planning and execution could potentially take place.

Another example of possible integration of the framework with a planning component comes from the author's collaboration in the context of the 'RELATE ITN' project, which sets as its goal engineering and provisioning of service-based cloud applications. Accordingly, work of Chatziprimou et al. (2013) investigates how utilisation of cloud resources can be optimised with respect to multiple cri-

teria. Such a multi-criteria optimisation is planned based on certain heuristics to achieve best possible results in the given context. In spite of being mainly IaaS-oriented, these planning capabilities can also be potentially applied to the PaaS segment and be integrated into our own EXCLAIM framework.

Machine learning techniques to be employed

As CAP environments are getting more and more complex, and extreme volumes of data are being generated at unpredictable rates, timely and prompt modification of the knowledge base becomes a major concern. Application of various Big Data analytics techniques to the domain of cloud monitoring and analysis came as a natural fit. These techniques, however, mainly addressed the volume and velocity of monitored data and can hardly help to cope with the variety and unpredictability of generated data.

When describing our approach, we relied on the assumption that the knowledge base is manually populated by the platform administrator and third-party service providers either at design-time or at run-time by updating the declaratively defined set of policies, so as to meet emerging requirements. In these circumstances, introduction of new policies or modification of existing ones implies presence of a human administrator. This requirement affects the overall capabilities of the framework to detect critical situations in a timely fashion, and contradicts with the very concept of autonomic computing.

This consideration suggests that the process of knowledge base maintenance can be potentially automated. In this light, application of various machine learning techniques has the potential to address this challenge [144]. With the help of machine learning, it would be possible to enable the knowledge base with self-training capabilities, so that new policies would be added and existing ones would be modified with respect to changing observations. The process of training a machine is typically two-step. First step – the supervised learning – involves training the system to predict values using various algorithms, and then matching predicted values with the actual results. This is an iterative process, which typically requires a lot of sample data to achieve a more precise model. Then, at the second step, the trained system is able to predict values (and keep on training). Ideally, the goal is to develop a self-training analysis engine which would be capable of adapting its knowledge base (i.e., C-SPARQL queries) to the changing context. In its simplest form, the self-training analysis component, based on historical observations, would accordingly adjust critical thresholds for particular SLA metrics, so as to make them more accurate and up-to-date.

Publishing results on-line as Linked Data

The original motivation behind employing the Semantic Web stack in the context of the presented research work was two-fold. First, we needed to move away from rigid, hard-wired definition of the knowledge base and associated diagnosis policies, and use a flexible, extensible, and platform-independent way of expressing knowledge. As a result, the EXCLAIM framework offers CAP providers an opportunity to define policies in a declarative and human-readable manner by using the underlying CSO as a common vocabulary of terms.

However, with this approach, CAP providers are expected to be responsible for maintaining the collection of policies, which concerns not only the internal platform components and native services, but also third-party add-on services, which are registered with the CAP marketplace. This, however, is typically not the case. CAP providers are not necessarily aware of the internal organisation of a third-party service and their behaviour; they treat them as black boxes and are not in a position to handle their adaptive behaviour. In these circumstances, third-party add-on providers, having registered and offered their software services in the cloud marketplace, are also expected to be responsible for customer support as far as their add-on services are concerned. It means that some parts of the knowledge base have to be delegated to reliable third parties registered with the CAP.

To address this challenge, we have utilised Linked Data principles to create a two-tier ontological architecture, consisting of the core ontology (i.e., the CSO) and multiple linked extensions to it, which are intended to contain self-governance policies, applicable to individual services and components. The primary goal of Linked Data is to enable discovery and sharing of semantically-enriched data over the Web using standardised technologies, such as URIs and RDF (Bizer et al., 2009). Accordingly, in the context of the EXCLAIM framework, we applied same principles to split the inherently heavy-weight monolithic ontological knowledge base into multiple linked pieces.

It is worth noting that Linked Data primarily refers to publishing RDF datasets (or 'instance data'), rather than to OWL, RDFS and SWRL vocabularies.[1] In this light, to benefit from application of Linked Data principles even further – that is, to support data integration and discovery – one of the future steps is to publish historical sensor data, collected by the EXCLAIM framework, as homogenised RDF triples in public online repositories. Indeed, we are already logging RDF

[1]Nevertheless, there is ongoing research aimed specifically at linking, sharing and re-using the underlying schemas, not just the datasets themselves. See Linked Open Vocabularies (`http://lov.okfn.org/dataset/lov/`) for a representative example of this research initiative.

data for the purposes of 'post-mortem' analysis, performance testing, and pattern identification. As we expect the EXCLAIM framework to mature and develop, publishing this information as Linked Data will provide researchers with access to real-world performance measurements, and has the potential, for example, to facilitate comparison between different CAPs.

9.3 Researcher's view

To conclude the thesis, we are providing the author's personal view and impressions from the conducted research, obtained results and further opportunities. The PhD research turned out to be a challenging and exciting period, which involved several pivot points, at which the research had to change its initial direction before the final version of the thesis was submitted.

The initial idea was to employ ontologies as a way of modeling cloud infrastructures and rules to express diagnosis policies. At that point, the project was solely associated with static reasoning. However, as it proceeded the requirement for a streaming approach became more and more apparent. As a result, the SSW-inspired solution has been chosen as the primary direction for further research. Feeling enthusiastic about this new promising idea, there was an ambitious attempt to create a software framework, which would fully implement the MAPE-K loop – that is, including the Planning and Execution steps. At this point, the context and the application scope for the conducted research also became clear – instead of focusing on cloud computing at all three levels – i.e., IaaS, PaaS, and SaaS – the was a clear gap of insufficient governance capabilities specifically in the context of CAPs. As a result, the idea was to develop a completely autonomic framework for self-adaptations in CAPs. Soon it became clear that each of the MAPE-K activities is worth a PhD research in its own right, and admittedly it would be better to focus on a single specific step within the MAPE-K cycle instead of ending up being 'everywhere and nowhere'. As a result, the research explicitly focused on the monitoring activities with support for problem detection – this seemed to be most challenging, interesting and promising direction. Additionally, this is where the SSW techniques were expected to bring most benefits. Essentially, this is when the approach finally crystallised and remained unchanged until the thesis submission date.

It is worth noting that conceptually the EXCLAIM approach is not limited to the domain of cloud platforms, and has the potential to be applied to other problem domains where timely processing and interpretation of dynamically generated heterogeneous data is required – one of such domains is, undoubtedly, the

emerging IoT. At the moment, the author is exploring potential possibilities of applying the existing work and experience accumulated during the PhD studies to this trending domain, and feeling enthusiastic about initial findings.

Acronyms

AMQP Advanced Messaging Queue Protocol. 102, 120

aPaaS Application Platform-as-a-Service. 25, 27

API Application Programming Interface. 18, 19, 27, 30, 35, 36, 43, 56, 77, 83, 94, 100, 101, 104–106, 119, 120, 124–126, 159, 206, 212

AWS Amazon Web Services. 22, 43, 72, 73, 77, 162, 208, 213

BPEL Business Process Execution Language. 75, 80, 82

BPM Business Process Management. 5

C-SPARQL Continuous SPARQL. 90, 101, 104, 106–108, 114, 115, 121, 122, 131, 133, 135–138, 142–146, 151–154, 164, 165, 171, 172, 175, 193, 194

CADA Collect – Analyze – Decide – Act. 6

CAP Cloud Application Platform. 3–13, 17, 18, 27–30, 33, 34, 36, 40, 44–46, 67, 70, 76, 79, 82–88, 90, 92, 94–99, 111–116, 119–121, 129, 132, 133, 138, 139, 144, 146, 150–153, 155, 158–163, 166, 168, 170–173, 175–177, 193–195, 198, 199, 204

CEP Complex Event Processing. 37, 52, 64, 165

CLAMS Cross-Layer Multi-Cloud Application Monitoring-as-a-Service. 77, 78, 213

CLI Command Line Interface. 127

CPU Central Processing Unit. 24, 25, 35, 37, 42, 43, 68, 69, 73, 74, 79, 87, 143, 152, 158, 204, 209

CRM Customer Relationship Management. 28

CSO Cloud Sensor Ontology. 67, 99, 100, 104, 106, 108, 110–112, 116, 118, 120–122, 132, 133, 136, 137, 154, 157, 162, 163, 165, 169, 171, 176, 193, 194

CSV Comma-Separated Value. 152

DBMS Database Management System. 105

DL Description Logic. 110, 165

DNS Domain Name System. 199

EB exabyte. 46, 47

EC2 Elastic Compute Cloud. 22, 23

ECA Event-Condition-Action. 37

FTP File Transfer Protocol. 29, 198

GB gygabyte. 46, 126, 128, 151, 152

GIS Geographic Information System. 55

GPS Global Positioning System. 48, 49

GUI Graphical User Interface. 112, 128, 141

HPC High Performance Computing. 68

HTTP HyperText Transfer Protocol. 19, 74, 209

IaaS Infrastructure-as-a-Service. 5, 24, 28, 42, 68–70, 76–80, 82, 158, 168, 177, 211, 212

IDE Integrated Development Environment. 25, 29, 53, 99, 110, 198

IFP Information Flow Processing. 50

IoS Internet of Services. 2, 4, 17, 20, 27, 70

IoT Internet of Things. 56, 116, 178

ISP Internet Service Provider. 20

IT Information Technology. 1, 4, 17–20, 22, 24, 30, 31, 37–41, 46, 47, 55, 116

JAR Java Archive. 53

JSON JavaScript Object Notation. 70, 91, 102, 105, 106, 125, 151, 163, 206

KPI Key Performance Indicator. 204

LOD Linked Open Data. 157

MAPE Monitor-Analyse-Plan-Execute. 34, 39, 98, 119

MAPE-K Monitor-Analyse-Plan-Execute-Knowledge. 5, 6, 8, 12, 15, 17, 18, 32, 34, 40, 44, 75, 76, 78, 90, 98, 110, 168, 174, 177, 210, 212, 213

MB megabyte. 128, 143, 151, 152

MISURE Monitoring Infrastructure using Streams on an Ultra-scalable, near-Real time Engine. 76, 212

MonaLISA Monitoring Agents in a Large Integrated Services Architecture. 75, 209

N3 Notation3. 58, 59

NIST National Institute of Standards and Technology. 22, 24

OGC Open Geospatial Consortium. 55

OOC Object-Oriented Computing. 18

OS Operating System. 2, 24, 25, 204

OWL Web Ontology Language. 8, 57, 58, 60–62, 76, 85, 90–95, 100, 101, 107, 108, 110, 115, 121, 122, 131, 137, 143, 156–160, 163–165, 176, 195

PaaS Platform-as-a-Service. 1–3, 5, 12, 17, 18, 24–30, 32, 36, 44, 69–71, 76, 77, 79, 82, 83, 119, 153, 168, 177, 211–213

PC Personal Computer. 143, 144

QoS Quality of Service. 41, 69, 77, 79, 213, 214

R2RML RDB to RDF Mapping Language. 106

RAM Random-Access Memory. 126

RDBMS Relational Database Management System. 47, 104

RDF Resource Description Framework. 8, 46, 57–60, 63–65, 76, 90–92, 95, 99–102, 104, 106–108, 113, 114, 120–122, 130, 131, 133, 134, 136–145, 147, 151, 153, 156, 159, 162–166, 169, 170, 172–174, 176, 193, 194

RDFS RDF Schema. 58, 60, 61, 176

REST Representational State Transfer. 160

RFID Radio Frequency Identification. 56

SaaS Software-as-a-Service. 5, 24, 25, 28, 70, 82, 158, 177, 211, 212

SALMon Service Level Agreement Monitor. 75, 210

SBA Service-based Application. 18, 20, 74, 75, 78, 80, 86, 95–97, 134, 168, 204

SDK Software Development Kit. 100

SEERC South-East European Research Centre. 111, 174

SFTP Secure File Transfer Protocol. 199

SLA Service Level Agreement. 6, 41, 75, 76, 78, 79, 83, 126, 175, 206, 210–214

SMS Short Message Service. 96

SMTP Simple Mail Transfer Protocol. 74, 209

SNMP Simple Network Management Protocol. 77

SOA Service-Oriented Architecture. 18, 21, 40, 76, 94, 210, 211

SOAP Simple Object Access Protocol. 160

SOC Service-Oriented Computing. 2, 17, 19–21, 23, 26, 36, 40, 41, 44, 75, 169

SPARQL SPARQL Protocol and RDF Query Language. 58, 60, 63–65, 90–92, 99, 101, 122, 162, 164

SPL Streams Programming Language. 53

SQL Structured Query Language. 50, 60, 91, 104, 105, 127, 163

SSH Secure Shell. 26, 29, 199

SSL Secure Sockets Layer. 29, 199

SSN Semantic Sensor Network. 12, 46, 57, 58, 62–65, 85, 86, 90, 111, 114, 157, 169

SSW Semantic Sensor Web. 7, 8, 10, 45, 46, 57, 58, 62–65, 85, 90, 94, 156, 157, 159, 160, 169, 170, 177, 195

SVN Subversion. 29, 198

SWE Sensor Web Enablement. 7, 55–57, 63, 65, 159, 163

SWRL Semantic Web Rule Language. 8, 58, 61, 76, 85, 90, 92–95, 99, 100, 104, 107, 108, 114, 115, 117–122, 131, 132, 137, 138, 143, 152, 154, 156–158, 161, 164, 165, 171, 172, 176, 193–195, 212

TB terabyte. 47, 126

TCP Transmission Control Protocol. 74, 209

Turtle Terse RDF Triple Language. 58–60

UML Unified Modeling Language. 79, 214

URI Universal Resource Identifier. 56, 59, 95, 119, 137, 162, 176

URL Universal Resource Locator. 101

USB Universal Serial Bus. 46

VM Virtual Machine. 2, 37, 74

W3C World Wide Web Consortium. 58, 62, 65, 164

WSDL Web Service Description Language. 75

WSLA Web Service Level Agreement. 76, 211

XML Extensible Markup Language. 52, 57–59, 63, 70, 76–78, 89, 100, 101, 106, 159, 163, 206, 210, 211

ZB zettabyte. 46

References

Abelson, H. (1999), *Architects of the information society: Thirty-five years of the laboratory for computer science at MIT*, MIT Press.

Akyildiz, I., Su, W., Sankarasubramaniam, Y. and Cayirci, E. (2002*a*), 'A survey on sensor networks', *IEEE Communications Magazine* **40**(8), 102–114.

Akyildiz, I., Su, W., Sankarasubramaniam, Y. and Cayirci, E. (2002*b*), 'Wireless sensor networks: a survey', *Computer Networks* **38**(4), 393–422.

Alhamazani, K., Ranjan, R., Jayaraman, P. P., Mitra, K., Liu, C., Rabhi, F. A., Georgakopoulos, D. and Wang, L. (2015), 'Cross-layer multi-cloud real-time application QoS monitoring and benchmarking as-a-service framework', *CoRR* **abs/1502.00206**.
URL: *http://arxiv.org/abs/1502.00206*

Alhamazani, K., Ranjan, R., Mitra, K., Jayaraman, P., Huang, Z., Wang, L. and Rabhi, F. (2014), CLAMS: Cross-layer Multi-cloud Application Monitoring-as-a-Service Framework, *in* '2014 IEEE International Conference on Services Computing (SCC)', pp. 283–290.

Alin, C. M. (2015), 'Top 7 Open-Source JSON-Binding Providers Available Today'. Accessed on 20/04/2016.
URL: *http://www.developer.com/lang/jscript/top-7-open-source-json-binding-providers-available-today.html/*

Amazon Web Services, Inc. (2015), 'Developer Guide – Amazon CloudWatch'. Accessed on 20/04/2016.
URL: *http://docs.aws.amazon.com/AmazonCloudWatch/latest/DeveloperGuide/*

Ameller, D. and Franch, X. (2008), Service level agreement monitor (SALMon), *in* 'Composition-Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on', pp. 224–227.

Anicic, D., Rudolph, S., Fodor, P. and Stojanovic, N. (2010), 'Stream reasoning and complex event processing in ETALIS', *Semantic Web* .

Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., Lee, G., Patterson, D., Rabkin, A. and Stoica, I. (2010), 'A view of cloud computing', *Communications of the ACM* **53**(4), 50–58.

Atzori, L., Iera, A. and Morabito, G. (2010), 'The Internet of Things: A survey', *Computer Networks* **54**(15), 2787–2805.

Baldwin, C. Y. and Clark, K. B. (2000), *Design rules: The power of modularity*, Vol. 1, MIT press.

Baldwin, C. Y. and Clark, K. B. (2003), 'Managing in an age of modularity', *Managing in the Modular Age: Architectures, Networks, and Organizations* **149**.

Barbieri, D., Braga, D., Ceri, S., Della Valle, E. and Grossniklaus, M. (2009), C-SPARQL: SPARQL for continuous querying, *in* 'Proceedings of the 18th international conference on World wide web', WWW '09, ACM, New York, NY, USA, pp. 1061–1062.

Barbieri, D., Braga, D., Ceri, S., Della Valle, E. and Grossniklaus, M. (2010), Stream Reasoning: Where We Got So Far, *in* 'Proceedings of the 4th International Workshop on New Forms of Reasoning for the Semantic Web: Scalable and Dynamic (NeFoRS)'.

Barbieri, D., Braga, D., Ceri, S., Valle, E., Huang, Y., Tresp, V., Rettinger, A. and Wermser, H. (2010), 'Deductive and Inductive Stream Reasoning for Semantic Social Media Analytics', *Intelligent Systems, IEEE* **25**(6), 32 –41.

Baryannis, G., Garefalakis, P., Kritikos, K., Magoutis, K., Papaioannou, A., Plexousakis, D. and Zeginis, C. (2013), Lifecycle Management of Service-based Applications on Multi-clouds: A Research Roadmap, *in* 'Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds', MultiCloud '13, ACM, New York, NY, USA, pp. 13–20.

Bechmann, A. and Lomborg, S. (2014), *The Ubiquitous Internet: User and Industry Perspectives*, Vol. 25, Routledge.

Benbernou, S., Hacid, L., Kazhamiakin, R., Kecskemeti, G., Poizat, J. L., Silvestri, F., Uhlig, M. and Wetzstein, B. (2008), 'State of the art report, gap analysis of knowledge on principles, techniques and methodologies for monitoring and adaptation of SBAs', *S-Cube Deliverable PO. JRA–1.2* **1**.

Berners-Lee, T., Hendler, J., Lassila, O. et al. (2001), 'The semantic web', *Scientific american* **284**(5), 28–37.

Bizer, C., Heath, T. and Berners-Lee, T. (2009), 'Linked data – the story so far', *International journal on semantic web and information systems* **5**(3), 1–22.

Blair, G., Coulson, G. and Grace, P. (2004), Research directions in reflective middleware: the Lancaster experience, *in* 'Proceedings of the 3rd workshop on Adaptive and reflective middleware', ARM '04, ACM, New York, NY, USA, pp. 262–267.

Bondi, A. B. (2000), Characteristics of scalability and their impact on performance, *in* 'Proceedings of the 2nd international workshop on Software and performance', ACM, pp. 195–203.

Boniface, M., Nasser, B., Papay, J., Phillips, S., Servin, A., Yang, X., Zlatev, Z., Gogouvitis, S., Katsaros, G., Konstanteli, K., Kousiouris, G., Menychtas, A. and Kyriazis, D. (2010), Platform-as-a-Service Architecture for Real-Time Quality of Service Management in Clouds, *in* '2010 Fifth International Conference on Internet and Web Applications and Services (ICIW)', pp. 155–160.

Botts, M., Percivall, G., Reed, C. and Davidson, J. (2008), 'OGC® sensor web enablement: Overview and high level architecture', *GeoSensor networks* pp. 175–190.

Botts, M. and Robin, A. (2014), 'OGC® SensorML: Model and XML Encoding Standard'.

Brandic, I. (2009), Towards Self-Manageable Cloud Services, *in* 'Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International', Vol. 2, pp. 128–133.

Bratanis, K. (2012), 'Towards engineering multi-layer monitoring and adaptation of service-based applications', *Dept. Comp. Sci., Univ. Sheffield, UK, Tech. Rep. CS-12-04* .

Bratanis, K., Dranidis, D. and Simons, A. J. H. (2012), The Challenge of Engineering Multi-Layer Monitoring & Adaptation in Service-Based Applications, *in* 'Proceedings of the 7th Annual South East European Doctoral Student Conference', pp. 497–503.

Brazier, F., Kephart, J., Van Dyke Parunak, H. and Huhns, M. (2009), 'Agents and Service-Oriented Computing for Autonomic Computing: A Research Agenda', *IEEE Internet Computing* **13**(3), 82 –87.

Breskovic, I., Haas, C., Caton, S. and Brandic, I. (2011), Towards Self-Awareness in Cloud Markets: A Monitoring Methodology, *in* '2011 Ninth IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC)', pp. 81 –88.

Buxmann, P., Hess, T. and Ruggaber, R. (2009), 'Internet of Services', *Business & Information Systems Engineering* **1**(5), 341–342.

Buyya, R., Vecchiola, C. and Selvi, S. T. (2013), *Mastering cloud computing: foundations and applications programming*, Newnes.

Calbimonte, J.-P., Jeung, H., Corcho, O. and Aberer, K. (2012), 'Enabling Query Technologies for the Semantic Sensor Web', *International Journal On Semantic Web and Information Systems* .

Cavalcante, E., Batista, T., Bencomo, N. and Sawyer, P. (2015), Revisiting Goal-Oriented Models for Self-Aware Systems-of-Systems, *in* 'Autonomic Computing (ICAC), 2015 IEEE International Conference on', IEEE, pp. 231–234.

Chaiken, R., Jenkins, B., Larson, P.-\., Ramsey, B., Shakib, D., Weaver, S. and Zhou, J. (2008), 'SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets', *Proc. VLDB Endow.* **1**(2), 1265–1276.

Chatziprimou, K., Lano, K. and Zschaler, S. (2013), Runtime Infrastructure Optimisation in Cloud IaaS Structures, *in* '2013 IEEE 5th International Conference on Cloud Computing Technology and Science (CloudCom)', Vol. 1, pp. 687–692.

Chen, M., Mao, S. and Liu, Y. (2014), 'Big Data: A Survey', *Mobile networks and applications* **19**(2), 171–209.

Compton, M., Barnaghi, P., Bermudez, L., Garcia-Castro, R., Corcho, O., Cox, S., Graybeal, J., Hauswirth, M., Henson, C., Herzog, A., Huang, V., Janowicz, K., Kelsey, W. D., Le Phuoc, D., Lefort, L., Leggieri, M., Neuhaus, H., Nikolov, A., Page, K., Passant, A., Sheth, A. and Taylor, K. (2012), 'The SSN ontology of the W3C semantic sensor network incubator group', *Web Semantics: Science, Services and Agents on the World Wide Web* **17**, 25–32.

Compton, M., Henson, C. A., Neuhaus, H., Lefort, L. and Sheth, A. P. (2009), A Survey of the Semantic Specification of Sensors., *in* 'SSN', pp. 17–32.

Cox, M. and Ellsworth, D. (1997), Application-controlled demand paging for out-of-core visualization, *in* 'Proceedings of the 8th conference on Visualization'97', IEEE Computer Society Press.

Cross, V. (2004), Fuzzy semantic distance measures between ontological concepts, *in* '2004 IEEE Annual Meeting of the Fuzzy Information Processing NAFIPS'04.', Vol. 2, pp. 635–640 Vol.2.

Curbera, F., Duftler, M., Khalaf, R., Nagy, W., Mukhi, N. and Weerawarana, S. (2002), 'Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI', *IEEE Internet Computing* **6**(2), 86–93.

Dautov, R., Paraskakis, I. and Kourtesis, D. (2012), An ontology-driven approach to self-management in cloud application platforms, *in* 'Proceedings of the 7th South East European Doctoral Student Conference (DSC 2012)', Thessaloniki, Greece, pp. 539–550.

Dautov, R., Paraskakis, I., Kourtesis, D. and Stannett, M. (2013), Addressing Self-Management in Cloud Platforms: a Semantic Sensor Web Approach, *in* 'Proceedings of the International Workshop on Hot Topics in Cloud Services (HotTopiCS 2013)', Prague, Czech Republic.

Dautov, R., Paraskakis, I. and Stannett, M. (2014*a*), 'Towards a Framework for Monitoring Cloud Application Platforms as Sensor Networks', *Cluster Computing* .

Dautov, R., Paraskakis, I. and Stannett, M. (2014*b*), 'Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms', *Journal of Cloud Computing* **3**(1), 1–12.

Dean, J. and Ghemawat, S. (2008), 'MapReduce: Simplified Data Processing on Large Clusters', *Commun. ACM* **51**(1), 107–113.

Delgado, N., Gates, A. Q. and Roach, S. (2004), 'A taxonomy and catalog of runtime software-fault monitoring tools', *IEEE Transactions on Software Engineering* **30**(12), 859–872.

Della Valle, E., Ceri, S., Barbieri, D., Braga, D. and Campi, A. (2009), A First Step Towards Stream Reasoning, *in* J. Domingue, D. Fensel and P. Traverso, eds, 'Future Internet', Vol. 5468 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, pp. 72–81.

Di, S., Kondo, D. and Cappello, F. (2013), Characterizing cloud applications on a Google data center, *in* 'Parallel Processing (ICPP), 2013 42nd International Conference on', IEEE, pp. 468–473.

Distefano, S., Merlino, G. and Puliafito, A. (2015), 'A utility paradigm for IoT: The sensing Cloud', *Pervasive and Mobile Computing* **20**, 127–144.

Dobson, S., Denazis, S., Fernandez, A., Gaiti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N. and Zambonelli, F. (2006), 'A survey of autonomic communications', *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* **1**(2), 223–259.

Ehlers, J., van Hoorn, A., Waller, J. and Hasselbring, W. (2011), Self-adaptive software system monitoring for performance anomaly localization, *in* 'Proceedings of the 8th ACM international conference on Autonomic computing', ACM, pp. 197–200.

Finos, R. (2015), 'Public Cloud Market Shares 2014 and 2015'. Accessed on 20/04/2016.
URL: *http://wikibon.com/public-cloud-market-shares-2014-and-2015/*

Galante, G. and de Bona, L. (2012), A Survey on Cloud Computing Elasticity, *in* '2012 IEEE Fifth International Conference on Utility and Cloud Computing (UCC)', pp. 263–270.

Ganek, A. G. and Corbi, T. A. (2003), 'The dawning of the autonomic computing era', *IBM Systems Journal* **42**(1), 5–18.

Gantz, J. and Reinsel, D. (2011), 'Extracting value from chaos', *IDC iView* **1142**, 1–12.

Gartner, Inc. (2015), 'APaaS – Application Platform as a Service'. Accessed on 20/04/2016.
URL: *http://www.gartner.com/it-glossary/application-platform-as-a-service-apaas/*

Gruber, T. R. (1993), 'A translation approach to portable ontology specifications', *Knowl. Acquis.* **5**(2), 199–220.

Gruber, T. R. (1995), 'Toward principles for the design of ontologies used for knowledge sharing', *International journal of human computer studies* pp. 907–928.

Harris, D. (2012), 'Heroku boss: 1.5M apps, many not in Ruby'. Accessed on 20/04/2016.
URL: *https://gigaom.com/2012/05/04/heroku-boss-1-5m-apps-many-not-in-ruby/*

Heineman, G. and Council, W. (2001), *Component-Based Software Engineering: Putting the Pieces Together*, Addison-Wesley.

Hellerstein, J. L. (2010), 'Google Cluster Data'. Accessed on 20/04/2016.
URL: *http://googleresearch.blogspot.com/2010/01/google-cluster-data.html/*

Hitzler, P., Krotzsch, M. and Rudolph, S. (2009), *Foundations of Semantic Web Technologies*, CRC Press.

Horn, P. (2001), 'Autonomic Computing: IBM's Perspective on the State of Information Technology', *Computing Systems* **15**(Jan), 1–40.

Huebscher, M. C. and McCann, J. A. (2008), 'A survey of autonomic computing – degrees, models, and applications', *ACM Comput. Surv* **40**(3), 1–28.

Huhns, M. and Singh, M. (2005), 'Service-oriented computing: key concepts and principles', *IEEE Internet Computing* **9**(1), 75–81.

IBM Corporation (2005), 'IBM Paves the Way for Mainstream Adoption of Autonomic Computing'. Accessed on 20/04/2016.
URL: *https://www-03.ibm.com/press/us/en/pressrelease/7623.wss/*

IBM Corporation (2013), 'IBM developerWorks : IBM InfoSphere Streams'. Accessed on 20/04/2016.
URL: *http://www.ibm.com/developerworks/bigdata/streams/*

IBM Corporation (2015), 'Four Vs of Big Data'. Accessed on 20/04/2016.
URL: *http://www.ibmbigdatahub.com/infographic/four-vs-big-data/*

Jain, V. and Singh, M. (2013), 'Ontology Development and Query Retrieval using Protege Tool', *International Journal of Intelligent Systems and Applications* **5**(9), 67–75.

Katsaros, G., Gallizo, G., Kübert, R., Wang, T., Fitó, J. O. and Henriksson, D. (2011), A multi-level architecture for collecting and managing monitoring information in cloud environments., *in* 'CLOSER', pp. 232–239.

Katsaros, G., Kousiouris, G., Gogouvitis, S. V., Kyriazis, D., Menychtas, A. and Varvarigou, T. (2012), 'A Self-adaptive hierarchical monitoring mechanism for Clouds', *Journal of Systems and Software* **85**(5), 1029–1041.

Kazhamiakin, R., Benbernou, S., Baresi, L., Plebani, P., Uhlig, M. and Barais, O. (2010), Adaptation of Service-Based Systems, *in* M. P. Papazoglou, K. Pohl, M. Parkin and A. Metzger, eds, 'Service Research Challenges and Solutions for the Future Internet', number 6500 *in* 'Lecture Notes in Computer Science', Springer Berlin Heidelberg.

Kazhamiakin, R., Pistore, M. and Zengin, A. (2010), Cross-Layer Adaptation and Monitoring of Service-Based Applications, *in* A. Dan, F. Gittler and

F. Toumani, eds, 'Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops', number 6275 *in* 'Lecture Notes in Computer Science', Springer Berlin Heidelberg, pp. 325–334.

Keller, A. and Ludwig, H. (2003), 'The WSLA framework: Specifying and monitoring service level agreements for web services', *Journal of Network and Systems Management* **11**(1), 57–81.

Kephart, J. and Chess, D. (2003), 'The vision of autonomic computing', *Computer* **36**(1), 41– 50.

Kephart, J., Chess, D., Boutilier, C., Das, R. and Walsh, W. (2007), 'An architectural blueprint for autonomic computing', *IEEE Internet Computing* **18**(21).

Kourtesis, D. (2011), Towards an Ontology-driven Governance Framework for Cloud Application Platforms, SEERC Technical Reports, South-East European Research Centre (SEERC), Thessaloniki, Greece.

Kourtesis, D., Bratanis, K., Bibikas, D. and Paraskakis, I. (2012), Software co-development in the era of cloud application platforms and ecosystems: The case of cast, *in* L. Camarinha-Matos, L. Xu and H. Afsarmanesh, eds, 'Collaborative Networks in the Internet of Services', Vol. 380 of *IFIP Advances in Information and Communication Technology*, Springer Berlin Heidelberg, pp. 196–204.

Lanzanasto, N., Komazec, S. and Toma, I. (2012), 'Reasoning over real time data streams', *ENVISION Deliverable* **D4.8**.

Le-Phuoc, D., Dao-Tran, M., Parreira, J. X. and Hauswirth, M. (2011), A native and adaptive approach for unified processing of linked streams and linked data, *in* 'Proceedings of the 10th international conference on The semantic web - Volume Part I', ISWC'11, Springer-Verlag, Berlin, Heidelberg, pp. 370–388.

Legrand, I., Newman, H., Voicu, R., Cirstoiu, C., Grigoras, C., Dobre, C., Muraru, A., Costan, A., Dediu, M. and Stratan, C. (2009), 'MonALISA: An agent based, dynamic service system to monitor, control and optimize distributed systems', *Computer Physics Communications* **180**(12), 2472–2498.

Leijon, V., Wallin, S. and Ehnmark, J. (2008), SALmon – A Service Modeling Language and Monitoring Engine, *in* 'IEEE International Symposium on Service-Oriented System Engineering SOSE'08', IEEE, pp. 202–207.

Lewis, J. and Fowler, M. (2015), 'Microservices'. Accessed on 20/04/2016.
URL: *http://martinfowler.com/articles/microservices.html/*

Liang, S. H., Croitoru, A. and Tao, C. V. (2005), 'A distributed geospatial infrastructure for Sensor Web', *Computers & Geosciences* **31**(2), 221–231.

Lupp, M. (2008), Open Geospatial Consortium, *in* 'Encyclopedia of GIS', Springer US, pp. 815–815.

MacKenzie, C. M., Laskey, K., McCabe, F. and Brown, P. F. (2006), Reference model for service oriented architecture 1.0, Technical report, OASIS Open.

Mahbub, K. and Spanoudakis, G. (2004), A Framework for Requirements Monitoring of Service Based Systems, *in* 'Proceedings of the 2nd International Conference on Service Oriented Computing ICSOC'04', ACM, New York, NY, USA, pp. 84–93.

Mahowald, R. P., Olofson, C. W., Ballou, M.-C., Fleming, M. and Hilwa, A. (2013), Worldwide Competitive Public Platform as a Service 2013–2017 Forecast, Technical Report 243315, IDC.

Manyika, J., Chui, M., Brown, B., Bughin, J., Dobbs, R., Roxburgh, C. and Byers, A. H. (2011), Big data: The next frontier for innovation, competition, and productivity., Technical report, McKinsey Global Institute.

Marconi, A., Bucchiarone, A., Bratanis, K., Brogi, A., Cámara, J., Dranidis, D., Giese, H., Kazhamiakin, R., De Lemos, R., Marquezan, C. C. et al. (2012), Research challenges on multi-layer and mixed-initiative monitoring and adaptation for service-based systems, *in* 'Proceedings of the First International Workshop on European Software Services and Systems Research: Results and Challenges', IEEE Press, pp. 40–46.

Margara, A. and Cugola, G. (2011), Processing flows of information: from data stream to complex event processing, *in* 'Proceedings of the 5th ACM international conference on Distributed event-based system DEBS'11', ACM, New York, NY, USA, pp. 359–360.

Margara, A., Urbani, J., van Harmelen, F. and Bal, H. (2014), 'Streaming the Web: Reasoning over dynamic data', *Web Semantics: Science, Services and Agents on the World Wide Web* **25**, 24–44.

McAfee, A. and Brynjolfsson, E. (2012), 'Big data: the management revolution', *Harvard business review* **90**(10), 60–66.

Mell, P. and Grance, T. (2009), 'The NIST definition of cloud computing', *National Institute of Standards and Technology* **53**(6), 50.

Mellor, C. (2015), 'Are you ready for the 40-zettabyte year?'. Accessed on 20/04/2016.
URL: *http://www.theregister.co.uk/2012/12/10/idc_zettabyte_fest/*

Meng, S., Iyengar, A. K., Rouvellou, I., Liu, L., Lee, K., Palanisamy, B. and Tang, Y. (2012), Reliable State Monitoring in Cloud Datacenters, *in* '2012 IEEE 5th International Conference on Cloud Computing (CLOUD)', pp. 951–958.

Meng, S., Kashyap, S. R., Venkatramani, C. and Liu, L. (2009), Remo: Resource-aware application state monitoring for large-scale distributed systems, *in* '29th IEEE International Conference on Distributed Computing Systems ICDCS'09', IEEE, pp. 248–255.

Meng, S. and Liu, L. (2013), 'Enhanced Monitoring-as-a-Service for Effective Cloud Management', *IEEE Transactions on Computers* **62**(9), 1705–1720.

Microsoft Corporation (2015), 'Service Oriented Architecture (SOA)'. Accessed on 20/04/2016.
URL: *https://msdn.microsoft.com/en-us/library/bb833022.aspx/*

Mohamed, A. (2015), 'A history of cloud computing'. Accessed on 20/04/2016.
URL: *http://www.computerweekly.com/feature/A-history-of-cloud-computing/*

Mueller, C., Oriol, M., Rodriguez, M., Franch, X., Marco, J., Resinas, M. and others (2012), SALMonADA: A platform for monitoring and explaining violations of WS-agreement-compliant documents, *in* 'Proceedings of the 4th International Workshop on Principles of Engineering Service-Oriented Systems', IEEE Press, pp. 43–49.

Muller, H. (2006), Bits of History, Challenges for the Future and Autonomic Computing Technology, *in* '13th Working Conference on Reverse Engineering, 2006. WCRE '06', pp. 9–18.

Nakamura, L., Estrella, J., Santana, R., Santana, M. and Reiff-Marganiec, S. (2014), A semantic approach for efficient and customized management of iaas resources, *in* '2014 10th International Conference on Network and Service Management (CNSM)', pp. 360–363.

Nami, M. and Sharifi, M. (2007), 'A survey of autonomic computing systems', *Intelligent Information Processing III* pp. 101–110.

Natis, Y. V., Knipp, E., Valdes, R., Cearley, D. W. and Sholler, D. (2009), Who's Who in Application Platforms for Cloud Computing: The Cloud Specialists, Technical report, Gartner Research.

Newman, H., Legrand, I., Galvez, P., Voicu, R. and Cirstoiu, C. (2003), 'Monalisa : A distributed monitoring service architecture', *CoRR* **cs.DC/0306096**.
URL: *http://arxiv.org/abs/cs.DC/0306096*

Newman, S. (2015), *Building Microservices*, O'Reilly Media.

Open Geospatial Consortium, Inc. (2015), 'Sensor Web Enablement'. Accessed on 20/04/2016.
URL: *http://www.ogcnetwork.net/SWE/*

Oracle Corporation (2015), 'Fast Data Solutions'. Accessed on 20/04/2016.
URL: *http://www.oracle.com/us/solutions/fastdata/index.html/*

Oriol, M., Marco, J., Franch, X. and Ameller, D. (2009), Monitoring adaptable SOA systems using SALMon, *in* 'Workshop on Service Monitoring, Adaptation and Beyond'.

Papazoglou, M., Pohl, K., Parkin, M., Metzger, A., Kazhamiakin, R., Benbernou, S., Baresi, L., Plebani, P., Uhlig, M. and Barais, O. (2010), *Service Research Challenges and Solutions for the Future Internet*, Vol. 6500 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg.

Papazoglou, M., Traverso, P., Dustdar, S. and Leymann, F. (2003), 'Service-oriented computing', *Communications of the ACM* **46**, 25–28.

Parkhill, D. F. (1966), *The challenge of the computer utility*, Vol. 2, Addison-Wesley Publishing Company.

Patel, P., Ranabahu, A. and Sheth, A. (2009), Service level agreement in cloud computing, *in* 'OOPSLA Cloud Computing workshop'.

Pettey, C. and van der Meulen, R. (2015), 'Gartner Says Worldwide Platform as a Service Revenue Is on Pace to Reach $1.2 Billion'. Accessed on 20/04/2016.
URL: *http://www.gartner.com/newsroom/id/2242415/*

Pokorny, J. (2013), 'NoSQL databases: a step to database scalability in web environment', *International Journal of Web Information Systems* **9**(1), 69–82.

Rackspace US, Inc. (2015), 'Cloud Monitoring – Server, App & Website Monitoring by Rackspace'. Accessed on 20/04/2016.
URL: *http://www.rackspace.com/cloud/monitoring/*

Ross, J. and Westerman, G. (2004), 'Preparing for utility computing: The role of IT architecture and relationship management', *IBM Systems Journal* **43**(1), 5–19.

Roy, J. and Ramanujan, A. (2001), 'Understanding Web services', *IT Professional* **3**(6), 69–73.

Rubin, D. L., Shah, N. H. and Noy, N. F. (2008), 'Biomedical ontologies: a functional perspective', *Briefings in Bioinformatics* **9**(1), 75–90.

Rumbaugh, J., Eddy, F., Lorensen, W., Blaha, M. and Premerlani, W. (1991), 'Object-oriented modeling and design'. Accessed on 20/04/2016.

Russell, S., Norvig, P. and Davis, E. (2010), *Artificial intelligence: a modern approach*, Prentice Hall – Upper Saddle River, NJ.

Russomanno, D., Kothari, C. and Thomas, O. (2005), Building a Sensor Ontology: A Practical Approach Leveraging ISO and OGC Models, *in* 'The 2005 International Conference on Artificial Intelligence', Press, pp. 637–643.

Rymer, J. R. and Ried, S. (2011), The Forrester Wave^{TM}: Platform-As-A-Service For Vendor Strategy Professionals, Q2 2011, Technical report, Forrester.

Salehie, M. and Tahvildari, L. (2009), 'Self-adaptive software: Landscape and research challenges', *ACM Transactions on Autonomous and Adaptive Systems (TAAS)* **4**(2), 14.

Schlenoff, C., Hong, T., Liu, C., Eastman, R. and Foufou, S. (2013), A literature review of sensor ontologies for manufacturing applications, *in* '2013 IEEE International Symposium on Robotic and Sensors Environments (ROSE)', pp. 96–101.

Schroth, C. and Janner, T. (2007), 'Web 2.0 and SOA: Converging Concepts Enabling the Internet of Services', *IT Professional* **9**(3), 36–41.

Sheth, A., Henson, C. and Sahoo, S. S. (2008), 'Semantic sensor web', *Internet Computing, IEEE* **12**(4), 78–83.

Smit, M., Simmons, B. and Litoiu, M. (2013), 'Distributed, application-level monitoring for heterogeneous clouds using stream processing', *Future Generation Computer Systems* **29**(8), 2103–2114.

Spanoudakis, G. and Mahbub, K. (2006), 'Non-intrusive monitoring of service-based systems', *International Journal of Cooperative Information Systems* **15**(03), 325–358.

Stevenson, G., Knox, S., Dobson, S. and Nixon, P. (2009), Ontonym: a collection of upper ontologies for developing pervasive systems, *in* 'Proceedings of the 1st Workshop on Context, Information and Ontologies', ACM.

Studer, R., Benjamins, V. and Fensel, D. (1998), 'Knowledge engineering: Principles and methods', *Data & Knowledge Engineering* **25**(1–2), 161–197.

Trihinas, D., Pallis, G. and Dikaiakos, M. (2014), JCatascopia: monitoring elastically adaptive applications in the cloud, *in* '14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014)', IEEE, pp. 226–235.

Tudorache, T., Noy, N. F., Tu, S. and Musen, M. A. (2008), Supporting Collaborative Ontology Development in Protege, *in* A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin and K. Thirunarayan, eds, 'The Semantic Web - ISWC 2008', number 5318 *in* 'Lecture Notes in Computer Science', Springer Berlin Heidelberg, pp. 17–32.

Urbani, J. (2010), Scalable and parallel reasoning in the Semantic Web, *in* 'The Semantic Web: Research and Applications', Springer, pp. 488–492.

Uschold, M. and Gruninger, M. (1996), 'Ontologies: Principles, methods and applications', *The knowledge engineering review* **11**(02), 93–136.

Vaquero, L. M., Rodero-Merino, L., Caceres, J. and Lindner, M. (2008), 'A break in the clouds: towards a cloud definition', *ACM SIGCOMM Computer Communication Review* **39**(1), 50–55.

Wähner, K. (2015), 'Real-Time Stream Processing as Game Changer in a Big Data World with Hadoop and Data Warehouse'. Accessed on 20/04/2016.
URL: *http://www.infoq.com/articles/stream-processing-hadoop/*

Wei, Y. and Blake, M. (2010), 'Service-Oriented Computing and Cloud Computing: Challenges and Opportunities', *IEEE Internet Computing* **14**(6), 72–75.

YouTube, LLC (2015), 'YouTube statistics'. Accessed on 20/04/2016.

Appendix A

List of the author's publications

- R. Dautov, I. Paraskakis, and M. Stannett. Utilising Stream Reasoning Techniques to Create a Self-Adaptation Framework for Cloud Environments. In *Proceedings of Sheffield University Engineering Symposium (SUSE 2015)*, 2015.

  Summary: This is an extended abstract, which contains a high-level overview of the EXCLAIM approach. It introduces the motivation behind the research, and briefly describes the EXCLAIM framework in two pages.

- R. Dautov, I. Paraskakis, M. Stannett. A monitoring and analysis framework to support self-management in cloud application platforms (Poster). In *Sheffield University Engineering Symposium (SUSE 2015)*, 2015.

  Summary: This is a poster submission, which contains a high-level overview of the EXCLAIM approach. It introduces the motivation behind the research, and briefly describes the EXCLAIM framework in two pages.

- K. Hamadache, P. Zerva, A. Polyviou, V. Simko, R. Dautov, F. Gonidis, and I. Paez Anaya. Cost in the Cloud Rationalisation and Research Trails. In *Proceedings of Second International Conference on Advanced Cloud and Big Data (CBD 2014)*, 2014.

  Summary: This is a joint paper, written in collaboration with 'RELATE' fellows. It is a position paper, which discusses possible ways of optimising energy consumption in cloud data centres using novel non-trivial approaches. For example, to benefit from night-time electricity tariffs, it might be possible to send computations across the globe to a location, which is currently night time. Same approach may be applied to executing cloud computations in locations, where it is currently winter, thereby saving expenses on cooling. The paper is not directly relevant to the PhD research presented in this thesis.

- R. Dautov, I. Paraskakis, M. Stannett. An Autonomic Semantic Framework for Cloud Application Platforms (Poster). In *5th International Conference on Knowledge Engineering and the Semantic Web (KESW 2014)*, 2014.

  Summary: This is a poster submission, which describes how the Semantic Web technologies have been applied in the context of the EXCLAIM framework. It highlights the role of the CSO and explains how it was used to define RDF streams, C-SPARQL queries, and SWRL policies.

- R. Dautov, I. Paraskakis, and M. Stannett. Cloud Sensor Ontology and Linked Data to Support Autonomicity in Cloud Application Platforms. In *Knowledge Engineering and the Semantic Web (KESW 2014)*, 2014. – Best paper award.

  Summary: This paper focuses on the CSO, which is used in the context of developing a monitoring and analysis framework for cloud platforms. First, the paper introduces the context of service-based cloud environments, and proceeds with the motivation for applying Semantic Web techniques. Then the paper describes individual classes and explains how they represent the cloud context. The paper received one of the three best paper awards.

- R. Dautov, I. Paraskakis, M. Stannett. Big Data Solutions for Cloud Application Platforms. In *Proceedings of the 9th Annual South East European Doctoral Student Conference (DSC 2014)*, 2014.

  Summary: This is a short paper, which introduces the challenge of processing extreme amounts of data in the context of CAPs when performing data monitoring and analysis. As a potential solution to overcome this problem, the paper presents IBM InfoSphere Streams and explains how it can be utilised to enable parallel processing of multiple data streams so as to achieve better performance.

- R. Dautov, I. Paraskakis, and M. Stannett. Towards a framework for monitoring cloud application platforms as sensor networks. *Cluster Computing*, 17(4):1203–1213, 2014.

  Summary: This is an extended version of a workshop submission, which elaborates on the novel interpretation of CAPs as networks of distributed software sensors. To justify this interpretation, the paper draws parallels between existing challenges associated with data monitoring in service-based cloud environments and physical sensor networks.

- R. Dautov, I. Paraskakis, and M. Stannett. Utilising Semantic Web technologies and Stream Reasoning to create an autonomic framework for cloud ap-

plication platforms (Poster). In *Doctoral Consortium of the 8th International Conference on Web Reasoning and Rule Systems (RR2014)*, 2014.

Summary: This is a poster submission, which describes how the Semantic Web technologies have been applied in the context of the EXCLAIM framework. It highlights the role of the CSO and explains how it was used to define RDF streams, C-SPARQL queries, and SWRL policies. The poster also focuses on RDF stream processing technology as a way of processing data in a dynamic and timely manner.

- R. Dautov, I. Paraskakis, and M. Stannett. Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms. *Journal of Cloud Computing*, 3(1):1-12, 2014.

  Summary: This is an extended version of a workshop submission, which discusses the potential of the Semantic Web stack to be applied to create a data monitoring framework for CAPs. The paper presents the architecture of the EXCLAIM framework, and highlights the role of the RDF stream processing in enabling on-the-fly data processing and overcoming data heterogeneity.

- R. Dautov, I. Paraskakis, and M. Stannett. Utilising Stream Reasoning Techniques to Create a Self-Adaptation Framework for Cloud Environments. In *Proceedings of 2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing (UCC)*, 2013.

  Summary: This paper introduces the emerging domain of stream reasoning (i.e., RDF stream processing) and explains how an existing implementation C-SPARQL was used to create an early version of the EXCLAIM framework to enable data monitoring and analysis in CAPs. An extended version of this paper was invited to be submitted to the Journal of Cloud Computing.

- R. Dautov, M. Stannett, and I. Paraskakis. On the role of stream reasoning in run-time monitoring and analysis in autonomic systems. In *Proceedings of the 8th South East European Doctoral Student Conference (DSC 2013)*, 2013.

  Summary: This is a short paper, which discusses the potential of the emerging stream reasoning research to address the challenge of monitoring and analysis in complex heterogeneous systems, such as CAPs. The paper argues that by applying stream reasoning techniques it is possible to address the heterogeneity and dynamic nature of monitored data.

- R. Dautov and I. Paraskakis. A vision for monitoring cloud application platforms as sensor networks. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference*, 2013.

Summary: This paper introduces the novel concept of treating CAPs as networks of distributed software sensors. The paper draws parallels between the problem domain of sensor networks and service-based cloud platforms, and argues that existing solutions can be re-used. An extended version of this paper was invited to be submitted to the Cluster Computing journal.

- R. Dautov, I. Paraskakis, D. Kourtesis, and M. Stannett. Addressing Self-Management in Cloud Platforms: a Semantic Sensor Web Approach. In *Proceedings of the International Workshop on Hot Topics in Cloud Services (HotTopiCS 2013)*, 2013.

  Summary: In this paper, the idea of re-using techniques from the SSW domain were proposed for the first time. The paper presents an early conceptual design of the EXCLAIM framework and explains how various SSW components can be used to implement elements of the framework.

- R. Dautov. An ontology-driven approach to self-management in cloud application platforms (Poster). In *9th Summer School on Ontology Engineering and the Semantic Web (SSSW'12)*, 2012. – Best poster award.

  Summary: This is a poster submission, which briefs on the potential of OWL ontologies to be used to define self-adaptation policies in complex cloud environments. The poster received one of the three best poster awards.

- R. Dautov, I. Paraskakis, and D. Kourtesis. An ontology-driven approach to self-management in cloud application platforms. In *Proceedings of the 7th South East European Doctoral Student Conference (DSC 2012)*, 2012. – Best paper award.

  Summary: This paper introduced the idea of using OWL ontologies and SWRL rules to be utilised to define self-adaptation policies in the context of cloud platforms, and distills benefits of doing so – namely, separation of concerns, declarative definition, human-readability, and increased opportunities for reliability, reuse and automation. It also introduces the notion of CAPs as the main context of the presented research. The paper received one of the three best paper awards.

Appendix B

Cloud Sensor Ontology

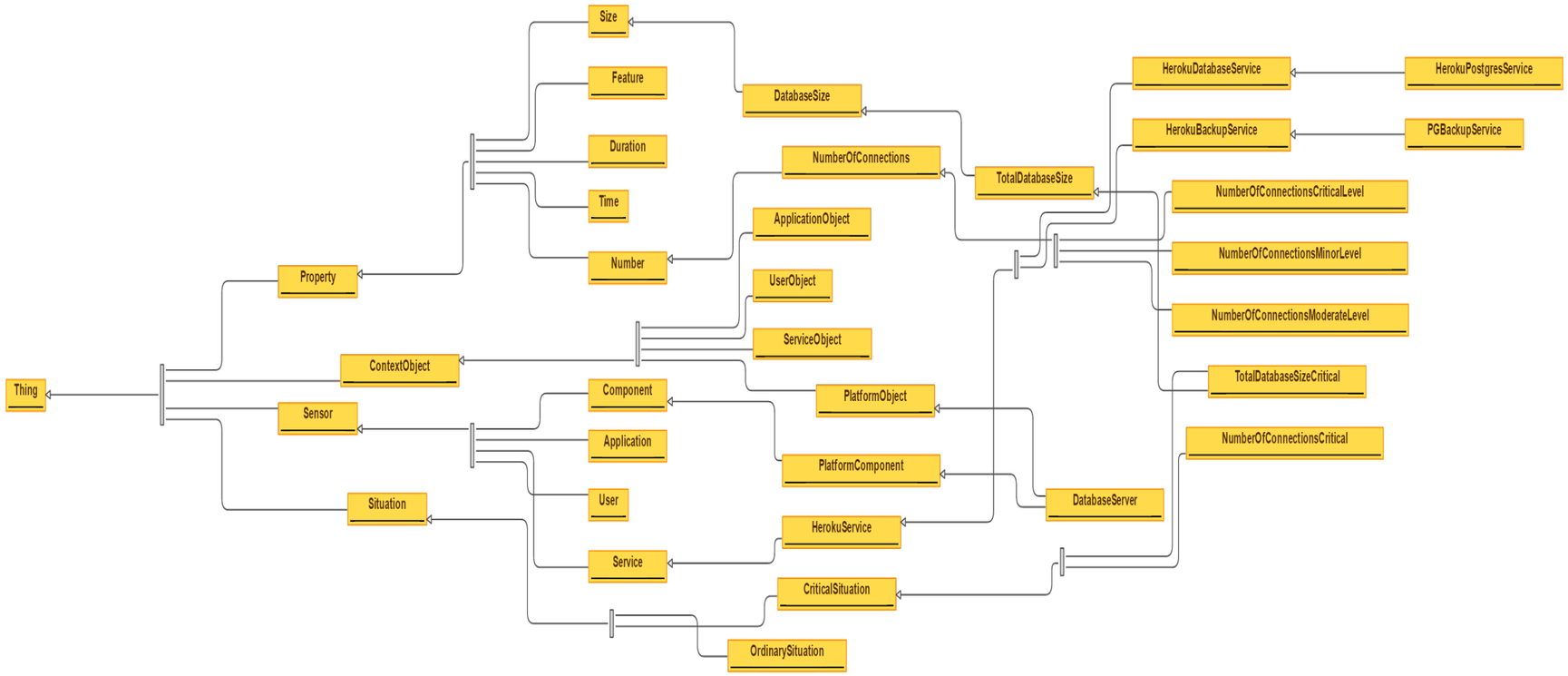

Figure B.1: Cloud Sensor Ontology used for the Destinator use case.

Appendix C

Survey of Cloud Application Platforms

In this appendix we are presenting a brief survey of existing cloud platform offerings (as of April 2016), which were examined with a goal to identify CAPs among them. Thus, we stress on the importance and relevance of the research question being addressed in this thesis, and, at the same, identify a potential application scope for our research.

Primary criteria for classifying a particular PaaS offering as a CAP were the presence of a cloud service marketplace and support for multiple programming languages and frameworks. When surveying cloud platforms, we also looked at the support for the software development process – that is, what tools and services they offer so as to enable faster software development and shorter time to market. In this light, the typical set of offered features includes the following:

- Web management console for accessing, deploying, managing, and controlling applications via a Web browser
- APIs for accessing, deploying, managing, and controlling applications programatically
- built-in data storage solutions
- software versioning (SVN, Git, Mercurial, Team Foundation, etc.)
- file transfers over the FTP connection
- IDE development plugins (e.g., for Eclipse IDE or Visual Studio) and Software Development Kits, which enable deploying applications straight from the IDE

- encryption and access management (e.g., SSL, SSH, Secure File Transfer Protocol (SFTP)) when connecting to the cloud platform
- simple application health monitoring and notification mechanisms, etc.

More sophisticated features also include support for Domain Name System (DNS) and email hosting, CRON job scheduling, 'point-and-click' interface for creating applications in a Web browser, etc. It is worth noting that the emergence and development of CAPs and cloud service marketplaces opens new opportunities for providing even richer support for software developers. Accordingly, even if some of the described features are not part of a PaaS offering initially (i.e., these features are not 'natively' supported), nevertheless, they might be offered to users as third-party services through the service marketplace.

Table C.1: Survey of Cloud Application Platforms.

PaaS offering	Programming languages	Service marketplace	Data storage	Hosting infrastructure	CAP
Anynines (http://www.anynines.com, founded in April 2013)	Java, Ruby, PHP, Python, Node.js, Go	7 services	PostgreSQL, MySQL, MongoDB, Redis	Private OpenStack infrastructure	**Yes**
Appfog (https://www.appfog.com, founded in August 2010)	Java, Python, Node, PHP, Ruby (+ 10 frameworks)	19 add-ons	ClearDB, Compose, ElephantSQL MongoDB and Redis	AWS (US, Europe, and Asia) and CenturyLink Cloud	**Yes**
AppHarbor (https://appharbor.com, founded in September 2010)	.NET	36 add-ons	Microsoft SQL Server, MySQL, MongoDB, RavenDB, Redis	AWS	**Yes**
AWS Elastic Beanstalk (https://aws.amazon.com/elasticbeanstalk, founded in January 2011)	Java, .NET, PHP, Node.js, Python, Ruby, and Go	N/a	Amazon RDS, Amazon DynamoDB, Amazon SimpleDB, MS SQL Server, Oracle, IBM DB2, Informix	AWS	**No**
Caspio (http://www.caspio.com, founded in 2000)	PHP, ASP and ASPX	13 extensions	MS SQL Server	N/a	**Yes**
CatN (https://catn.com)	PHP	N/a	MySQL	Private infrastructure	**No**
cloudControl (https://www.cloudcontrol.com)	Java, Python, Node, PHP, Ruby	38 add-ons	MySQL, PostgreSQL, MongoDB	AWS	**Yes**
Pivotal Cloud Foundry (http://pivotal.io/platform, founded in April 2011)	Java, Ruby, Python, Go, PHP or Node.js	13 services	MySQL, Redis, Cassandra, MongoDB, Neo4j	VMware, AWS, Openstack	**Yes**

Cloudways (http://www.cloudways.com)	PHP	9 add-ons	MySQL	AWS, DigitalOcean and Google Compute Engine	**Yes**
dotCloud (https://www.dotcloud.com)	Java, Python, Node, PHP, Ruby, Perl, Opa	11 services	MySQL, PostgreSQL, MongoDB, Redis	Google Compute Engine	**Yes**
Fortrabbit (http://www.fortrabbit.com, founded in January 2013)	PHP	N/a	MySQL	AWS	**No**
Google App Engine (https://appengine.google.com, founded in April 2008)	Java, Python, Go, PHP	36 features	App Engine Datastore, Google Cloud SQL (MySQL)	Google Compute Engine	**Yes**
Engine Yard (https://www.engineyard.com, founded in 2006)	PHP, Ruby, Node.js	64 add-ons	PostgreSQL, MySQL, MongoDB, Riak, SpacialDB	AWS, MS Azure	**Yes**
Heroku (https://www.heroku.com, founded in 2007)	Ruby, Node.js, Python, Java, and PHP	160 add-ons	MySQL, PostgreSQL, MongoDB, CouchDB (+ 12 data storage add-ons)	AWS	**Yes**
IBM Bluemix (http://www.ibm.com/cloud-computing/bluemix/, founded in June 2014)	Ruby, Node.js, Python, Java, Go, PHP	82 services	Microsoft SQL Server, MySQL, MongoDB, Redis, PostgreSQL	IBM SoftLayer Cloud	**Yes**
Jelastic (https://jelastic.com, founded in 2010)	Java, PHP, Python, Node.js, Ruby, .NET	N/a	MariaDB, MySQL, PostgreSQL, MongoDB, CouchDB	AWS, Google Compute Engine	**No**
webMethods AgileApps Cloud (http://www.agileappscloud.com)	Java	N/a	MySQL	N/a	**No**

Microsoft Windows Azure (`https://azure.microsoft.com`, founded in February 2010)	.NET, Java, Node.js, PHP, Python, Ruby	55 application services (+216 third party data services)	MS SQL Server, Oracle DB	MS Azure	**Yes**
Red Hat OpenShift (`https://www.openshift.com`, founded in May 2011)	Java, Ruby, PHP, Node.js, Python and Perl	29 add-ons	MySQL, PostgreSQL, MongoDB, MS SQL Server	AWS	**Yes**
OutSystems Platform (`http://www.outsystems.com/platform`, founded in 2001)	Java, .NET	200+ mobile services and connectors	PostgreSQL, MySQL, MongoDB, Redis	AWS, private infrastructure	**Yes**
Pagodabox (`https://pagodabox.io/`, founded in October 2010)	PHP	N/a	MySQL, PostgreSQL, MongoDB	Private infrastructure	**No**
Evia Cloud (`http://relbit.com`, founded in 2011)	PHP	9 add-ons	MySQL, MongoDB	Private infrastructure	**Yes**
Force.com (`http://force.com`, founded in 2005)	Apex	2500+ community apps in AppExchange	N/a	Salesforce.com infrastructure	**Yes**
Zoho Creator (`https://www.zoho.com/creator`, founded in 2005)	Deluge Script	N/a	N/a	N/a	**No**

Appendix D

Survey of the state of the art

A literature survey on monitoring and analysis in service-based cloud environments has identified a wide spectrum of existing approaches and tools, which distinguish between one another by a number of important characteristics. In order to provide a holistic, comprehensive, and integrated view on the existing body of research work, we will first present and describe main criteria, with respect to which we surveyed the literature. These criteria were distilled from several existing taxonomies, and are a result of thorough consideration and study of the state of the art in the considered problem domain. Namely, in our research, we relied on the following taxonomies for software monitoring and analysis:

- Adaptation taxonomy devised within the S-Cube project[1] (Kazhamiakin, Benbernou, Baresi, Plebani, Uhlig and Barais, 2010), which was developed to address the domain of monitoring and adaptation of SBAs
- Taxonomy of run-time software-fault monitoring tools by Delgado et al. (2004)
- Taxonomy of software self-adaptation by Salehie and Tahvildari (2009).

It has to be explained that the range of classification criteria, which could also be potentially included in this survey, is somewhat wider. Our goal is to provide a global, systematic and integrated view on the current state of the art, which inevitably requires omitting minor details. When distilling the classification criteria, we tried to consider them in connection with the problem domain of cloud, self-governance, data monitoring and analysis, as opposed to looking at them only from a generic software monitoring perspective. Next, we describe the main classification criteria, which will help us to systematise the existing body of

[1] http://www.s-cube-network.eu/

research work, outline existing research and technological gaps, and, accordingly, position our own work respectively.

Data collection and monitoring

These criteria are mainly related to how and from where monitored data is collected. Accordingly, it includes the following criteria:

Level of monitoring within CAPs SBAs deployed and running on a CAP span across four conceptual layers. Each of these layers provides its individual perspective on the execution of the deployed applications. This four-tier architecture consists of Service Infrastructure (SI), Application Performance (AP), Service Composition and Coordination (SCC), and Business Process Management (BPM) abstraction layers (Papazoglou et al., 2010). Accordingly, the SI monitoring targets at detecting failures in the underlying hardware (e.g., network, CPU, memory, etc.) and software (e.g., OS, virtualisation layer, etc.) infrastructures. The AP monitoring collects data about software performance at run-time – e.g., number of invocations, response rates, amount of inbound/outbound traffic, number of execution threads, etc. The SCC monitoring aims at detecting software failures due to violation of functional and non-functional constraints, resulting from interactions between individual services in complex service-based applications (Benbernou et al., 2008). This is particularly important in the context of CAPs, which are characterised with multiple add-on services, which are widely used to create complex service-based compositions. The BPM monitoring concentrates on business performance of SBAs in terms of Key Performance Indicators (KPIs) and checks the conformance with respect to a set of business policies. In recent years, more comprehensive approaches aiming at multi-layer monitoring of SBAs have been attracting a lot of attention as well (Kazhamiakin, Pistore and Zengin, 2010, Marconi et al., 2012).

Source of collected data Sources, which generate data for monitoring, comprise both monitored software deployed on a cloud platform and its operational context, which may include, for example, the execution environment, operating system, depending services and applications. Some forms of monitoring data sources include logs, events, messages, databases etc.

Data analysis

These criteria describe the process of data interpretation after it was collected. Among other things, analysis activities may include detecting critical situations, identifying adaptation requirements, choosing appropriate adaptation strategies, etc. We identify two main criteria in this respect:

Timeliness With respect to this criterion, monitoring mechanisms can be classified into proactive and reactive approaches. Proactive approaches aim at predicting potential failures of the monitored application before they occur, while reactive approaches detect failures after they have already happened. The latter type also includes techniques such as post-mortem analysis – i.e., offline interpretation of monitored values long after they have been observed.

Level of automation We can distinguish between three main forms of analysis with respect to the level of automation. To certain extent, this classification correlates with classification of degrees of self-management in computing systems, summarised in Table 2.1. Completely automated analysis is performed, based on a set of predefined rules without human intervention. The interactive analysis allows for human operators to supervise the process and take certain decisions. In the manual analysis there is no automation, and interpretation of observed values and problem diagnosis are solely performed by human operators, who are responsible for defining further adaptation requirements and choosing appropriate adaptation strategies.

Architecture details

These criteria describe design and implementation aspects of existing approaches, and include the following six:

Intrusiveness Depending on the usage of the data collection process, it can be either intrusive or non-intrusive. As opposed to the non-intrusive approach, in order to increase the visibility into the execution of cloud-hosted software, the intrusive approach assumes instrumenting monitored applications or their execution environment with probes – software components responsible for collecting and transferring monitored data.

Component distribution We can identify two main categories of existing tools for monitoring and analysis in service-based cloud environments. A centralised architecture assumes that all the monitoring and analysis mecha-

nisms are implemented within a single central unit. A decentralised architecture implies having several independent units for data processing and storage, which may even be deployed on separate execution platforms. Such approaches often rely on replication techniques, which help to avoid possible bottlenecks and failures, but might not provide a global view on the managed cloud environment.

Language type The knowledge base of a self-governance mechanism may include, for example, an architectural model of the managed environment, monitoring properties, diagnosis policies, constraints, assertions, etc. Here we distinguish between two main types of languages, which serve to specify and capture this knowledge – imperative and declarative approaches. The former refers to explicitly defining all the knowledge directly in the programming source code, and is known for its relative simplicity and faster execution at a price of rigidness (i.e., source code modifications lead to software recompilation, redeployment and restart). The latter approach, on the other hand, employs certain formalisms to declaratively define knowledge and decouple it from the actual programming code. With the declarative approach, policy definition is decoupled from the actual policy enforcement. Declarative definitions range from simple text file definitions (e.g., XML, JSON, etc.) to more sophisticated formalisms, such as various types of algebra, automata, and logic.

Extensibility This feature reflects how easy and transparently to the user new emerging metrics, aspects and policies can be integrated into (or removed from) the monitoring and analysis mechanism. Admittedly, it is not easy and straight-forward do decide if a particular approach is extensible enough or not. Potentially, we can argue that approaches using declarative languages to define the knowledge base are more likely to be classified as extensible, as opposed to imperative languages (e.g., Java or C). We also classify a particular approach as extensible, if authors discuss potential ways of extending their solutions to capture new metrics via, for example, an API, even it requires some hard-coding.

Perspective Various stakeholders can be concerned with governance processes, and three main perspectives can be identified in this respect (Bratanis, 2012). The customer perspective mainly concerns external monitoring, which aims at checking if cloud service providers deliver what has been agreed on in SLAs, since it is the main point of interest for a customer. The provider perspective concerns monitoring processes with the purpose of checking if

cloud-hosted software satisfies specific requirements – for example, if service response times are within allowed limits, or resources are properly shared between several applications. A monitoring and analysis process can also be performed by a third party, which provides an independent view on the managed cloud ecosystem.

Maturity level This criteria indicates the current development stage of a particular cloud self-governance tool and the overall maturity of a corresponding approach (Delgado et al., 2004). Typically, we can distinguish between two types – research prototypes and already established tools available for public use either as a commercial or a freeware product.

Table D.1: Survey of the state of the art in monitoring and analysis of service-based cloud platforms.

	Monitoring		Analysis		Architecture					
Approach	**Monitoring level**	**Data sources and metrics**	**Automation**	**Timeliness**	**Intrusiveness**	**Distribution**	**Language type**	**Extensibility**	**Perspective**	**Maturity level**
1. Platform-native built-in monitoring solutions										
Amazon CloudWatch is a generic monitoring service, which is able to monitor the whole stack of AWS products, including Elastic Beanstalk, and display it as charts and diagrams. It also allows defining user alerts and notifications.	SI, AP	Infrastructure metrics, software performance metrics	Automated (SI), interactive (AP)	Reactive	Non-intrusive	N/a	N/a	Customisable alerting and logging	Customer	Commercial product
Rackspace CloudKick is a generic monitoring service for the whole stack of Rackspace cloud offerings, which is able to monitoring standard infrastructure-oriented metrics. At its current state, it is rather simplistic, but allows user to define their own customised extensions to meet individual user requirements.	SI, AP	Infrastructure metrics, software performance metrics	Automated (SI), interactive (AP)	Reactive	Non-intrusive	N/a	N/a	Customisable alerting and logging; user-defined extension plugins	Customer	Commercial product
Google App Engine Dashboard is a simple monitoring solution offered by Google to provide users with information about the current status of their deployed applications and network statistics.	SI, AP	Software performance metrics	Manual, interactive	Reactive	Non-intrusive	N/a	N/a	N/a	Customer	Commercial product
2. Third-party monitoring frameworks										

Nagios is a stack of multi-purpose monitoring frameworks, which enable resource and application performance monitoring based on an extensible architecture. It offers various monitoring services such as monitoring of host resources (e.g., CPU/memory utilisation, response times, etc.), network services and protocols (e.g., SMTP, TCP, HTTP, etc.).	SI, AP	Infrastructure metrics, software performance metrics	Manual, interactive	Reactive	Non-intrusive	Distributed	C (back end), PHP (front end)	N/a	Customer	Commercial product
New Relic offers a whole stack of rich monitoring solutions, ranging from the level of data centre infrastructure resources to the level of individual applications and databases. A New Relic agent follows a non-intrusive approach to data collection and enables monitoring run-time application performance.	SI, AP	Infrastructure metrics, software performance metrics	Manual, interactive	Reactive	Non-intrusive	Distributed	C (back end), PHP (front end)	N/a	Customer	Commercial product
Zabbix (`http://www.zabbix.com/`) is an enterprise open-source monitoring solution, primarily focusing on networks and applications. It is build on a distributed architecture with a centralised web administration.	SI, AP	Infrastructure metrics	Manual, interactive	Reactive	Non-intrusive	Distributed	C (back end), PHP (front end)	N/a	Customer	Commercial product
MonaLISA (Newman et al., 2003, Legrand et al., 2009) aims to provide monitoring services to support control, optimisation, and management tasks in large-scale highly-distributed cloud systems. It primarily focuses on collecting data at the infrastructure level.	SI, AP	Infrastructure metrics	Manual, interactive	Reactive	Non-intrusive	Distributed	Java, C	N/a	Provider	Research prototype
Paraleap AzureWatch is a third-party monitoring service, which supports monitoring Azure-hosted applications, and offers support for data visualisation, logging and alerting.	SI, AP	Infrastructure metrics, software performance metrics	Manual, interactive	Non-intrusive	N/a	Centralised	N/a	Customisable alerting and logging	Customer	Commercial product

Heroku add-on services (Librato, New Relic APM, Hosted Graphite, Pingdom, Still Alive, Blackfire.io, Dead Man' Snitch, Vigil Monitoring Service, Rollbar, Sentry, Bugsnag, RuntimeError, Honeybadger, Appsignal, Exceptiontrap, Airbrake Bug Tracker, Raygan.io, Informant, RedisMonitor) offer a wide range of monitoring tools form simple 'heart beat monitoring' to more sophisticated support for analysing fine-grained performance data.	SI, AP	Software performance metrics, infrastructure metrics	Manual, interactive, automated	Reactive	Non-intrusive	Centralised	N/a	Customisable alerting and logging	Customer	Commercial product
Datadog (https://www.datadoghq.com/) is an application performance monitroing framework, which can be integrated with a range of cloud offerings (including Google App Engine and MS Azure). It can collect, aggregate, and visualise a wide range of AP metrics.	AP	Software performance metrics	Manual, interactive	Reactive	Non-intrusive	Centralised	N/a	Customisable alerting and logging	Customer	Commercial product
3. SOA solutions										
SALMon (Ameller and Franch, 2008, Oriol et al., 2009, Leijon et al., 2008, Mueller et al., 2012, Mahbub and Spanoudakis, 2004, Spanoudakis and Mahbub, 2006) aims at generic monitoring of heterogeneous web services, detecting SLA violations and taking corresponding adaptive decisions. It fully implements the MAPE-K loop and relies on a novel domain-specific language to express the analysis and diagnosis policies. The SALMon domain-specific language serves to model the environment and overcome heterogeneity in service descriptions, which makes it relevant and applicable to the PaaS domain. SALMon utilises a streaming approach to data processing to achieve timely reactions.	AP, SCC	SLAs in service compositions	Manual, interactive	Reactive	Non-intrusive	Distributed	XML-based domain specific language	N/a	Provider	Research prototype

The WSLA framework (Keller and Ludwig, 2003, Patel et al., 2009) is able to specify and monitor SLAs in SOA environments, including clouds. It employs an XML-based domain-specific language to define SLAs and constraints, which allows for extending the framework to cover all three layers of cloud computing.	SCC	SLAs in service compositions	Manual, interactive	Reactive	Non-intrusive	Centralised	XML-based domain-specific language	Can be extended to include cloud-related metrics	Provider	Research prototype
4. IaaS-oriented approaches										
Meng and Liu (2013), Meng et al. (2012, 2009) suggest a cloud monitoring framework offered as a service. In their paper, the authors consider an IaaS-oriented scenario, but claim that this Monitoring-as-a-Service solution potentially targets at all three levels of cloud computing. Main benefits of the proposed approach include lower monitoring cost, higher scalability, and better multitenancy performance	SI, AP	Infrastructure metrics, software performance metrics	Manual, Interactive	Reactive	Non-intrusive	Centralised	Java implementation	Can be potentially extended to PaaS and SaaS levels.	Provider	Research prototype
Katsaros et al. (2012, 2011) propose a cloud monitoring and self-adaptation mechanism, which spans across different levels of the IaaS level and collects monitoring data from application, virtual and physical infrastructure, and additionally considers the energy efficiency dimension. The authors also consider the PaaS segment as a an application scope for their research.	SI, AP	Infrastructure metrics, software performance metrics, energy efficiency metrics	Interactive	Reactive	Non-intrusive	Centralised	N/a	Can be potentially extended to the PaaS level	Provider	Research prototype

Nakamura et al. (2014) propose utilising Semantic Web technologies to support self-management at the IaaS level. The authors made created an ontology-based architectural model of a cloud data centre and delegate decision taking tasks to the SWRL reasoning engine. The proposed architecture implements the MAPE-K reference model with a goal to maintain established SLAs and optimise cloud resource consumption.	SI	Infrastructure metrics	Interactive, automated	Reactive	Non-intrusive	Centralised	Semantic Web languages	N/a	Provider	Research prototype
MISURE framework (Smit et al., 2013) builds the monitoring infrastructure on the stream processing technology (S4 and Storm). It is integrated with other existing infrastructure monitoring mechanisms to collect relevant metrics across a wide range of heterogeneous cloud environments. It follows a modular approach, so that custom extensions can be integrated into the framework.	SI, AP	Infrastructure metrics, software performance	Interactive, automated	Reactive	Non-intrusive	Centralised	Java	Custom extensions can be plugged into the framework to support data collection at various levels	Provider	Research prototype
JCatascopia (Trihinas et al., 2014) is an automated, multi-layer, interoperable framework for monitoring and managing elastic cloud environments. It primarily focuses on the IaaS segment of cloud computing, but also offers the Probe API to implement customisable extensions and metrics.	SI, AP	Infrastructure metrics, software performance metrics	Interactive, automated	Reactive	Non-intrusive by defult, but can be extended with probes	Distributed	Java	Can be extended to cover the PaaS and SaaS levels	Provider, customer	Research prototype

The CLAMS framework (Alhamazani et al., 2015, 2014) represents and effort to create a monitoring solution spanning across cloud computing layers and several platforms. The proposed architecture can be deployed on several cloud platforms (e.g., the authors validated it on AWS and MS Azure) and is capable of performing QoS monitoring of application components. As far as the PaaS is concerned, the CLAMS's monitoring capabilities are limited to monitoring databases and application containers.	SI, AP	Infrastructure metrics, software performance metrics	Manual, interactive	Reactive	Non-intrusive	Distributed (in a multi-cloud environment)	Java	N/a	Provider, customer	Research prototype
5. Truly PaaS-oriented approaches										
Brandic (2009) focuses on the platfrom level of cloud computing and describe how cloud services can be described using novel domain-specific languages, which focus on autonomic behaviour of a service. Based on these self-descriptive definitions, the proposed middleware platform is able to execute self-management within a cloud platform. The approach follows the MAPE-K loop, and claims to be applicable to arbitrary cloud services.	SI, AP, SCC	SLA metrics	Automated	Reactive	Non-intrusive	Centralised	Declarative	Can be potentially extended to cover a wider range of cloud service self-management aspects.	Provider	Research prototype
Breskovic et al. (2011) focus on cloud markets – a subset of cloud platforms which are used for deploying software markets. Their goal is to create a cloud market, which "has the ability to change, adapt or even redesign its anatomy and/or the underpinning infrastructure during runtime in order to improve its performance". To do so, the authors extend an existing data collection framework with cloud market-relevant sensors, which provide necessary data for performing analysis.	SI, AP, BPM	SLA metrics, cloud market metrics	Automated	Reactive	Intrusive	Centralised	Declarative	N/a	Provider	Research prototype

Boniface et al. (2010) focus on the glspaas level of cloud computing, aiming at developing an automated QoS management architecture. In their paper, the authors focus on a cloud platform hosting a wide range of multimedia applications, which are monitored at various levels using the proposed software framework. The proposed approach utilises several modeling techniques (e.g., neural networks, UML, etc.) to detect potential SLA violations and support QoS.	SI, AP, BPM	SLA metrics	Automated	Reactive	Intrusive	Centralised	UML, Artificial Neural Networks	N/a	Provider	Research prototype