Journal of Cloud Computing
a SpringerOpen Journal

RESEARCH Open Access

Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms

Rustem Dautov[1*], Iraklis Paraskakis[1] and Mike Stannett[2]

Abstract

As cloud application platforms (CAPs) are reaching the stage where the human effort required to maintain them at an operational level is unsupportable, one of the major challenges faced by the cloud providers is to develop appropriate mechanisms for run-time monitoring and adaptation, to prevent cloud application platforms from quickly dissolving into a non-reliable environment. In this context, the application of intelligent approaches to Autonomic Clouds may offer promising opportunities. In this paper we present an approach to providing cloud platforms with autonomic capabilities, utilising techniques from the Semantic Web and Stream Reasoning research fields. The main idea of this approach is to encode values, monitored within cloud application platforms, using Semantic Web languages, which then allows us to integrate semantically-enriched observation streams with static ontological knowledge and apply intelligent reasoning. Using such run-time reasoning capabilities, we have developed a conceptual architecture for an autonomous framework and describe a prototype solution we have constructed which implements this architecture. Our prototype is able to perform analysis and failure diagnosis, and suggest further adaptation actions. We report our experience in utilising the Stream Reasoning technique in this context as well as further challenges that arise out of our work.

Keywords: Cloud computing; Autonomic computing; Monitoring; Analysis; Stream reasoning

Introduction

Cloud computing impacts upon almost every aspect of daily life and the economy – pervasive cloud services are revolutionising the way we do business, maintain our health, and educate and entertain ourselves. Along with recent advances in computing, networking, software, hardware and mobile technologies, however, come emerging challenges to our ability to ensure that cyberspace resources and services are properly regulated, maintained and secured. The ubiquitous insertion of increasingly automated processes and procedures into traditional personal, scientific and business activities dictates a need to design such systems carefully, so as to guarantee that these associated challenges are properly met. Managing such large scale systems effectively inevitably means that resources will need to become increasingly "autonomous", capable of managing themselves – and cooperating with one another - without manual intervention.

In particular, the Platform-as-a-Service (PaaS) segment of cloud computing has been steadily growing over the past several years, with more and more software developers choosing cloud application platforms as convenient ecosystems for developing, deploying, testing and maintaining their software. Following the principles of Service-Oriented Computing (SOC), such platforms offer their subscribers a wide selection of pre-existing and reusable services, ready to be seamlessly integrated into users' applications. However, by offering such a flexible model for application development, in which software assets are assembled from existing components just like a Lego® construction set, cloud platform providers increasingly find themselves in a situation where the ever-growing complexity of entangled cloud environments poses new challenges as to how such systems should be monitored and managed.

*Correspondence: rdautov@seerc.org
[1] South-East European Research Centre, International Faculty of the University of Sheffield, City College, 24 Proxenou Koromila Street, 54646 Thessaloniki, Greece
Full list of author information is available at the end of the article

In this context, we present a novel approach to developing autonomic cloud application platforms, based on our vision of treating cloud platforms as sensor networks [1]. Our approach makes intelligent re-use of existing solution strategies and products (specifically, Stream Reasoning and the Semantic Web technology stack), to create a general-purpose autonomous framework. In this paper we consider how cloud application platform providers can benefit from our approach. As will be explained in more detail below, our approach relies on annotating monitored values with semantic descriptions, thereby enabling the framework to combine observation streams with static ontological knowledge and perform run-time formal reasoning. This in turn opens promising opportunities for performing run-time analysis, problem diagnosis, and suggesting further adaptation actions. We also discuss potential shortcomings of our approach and consider ways of overcoming them.

The rest of the paper is organised as follows. Section "Background and motivation" is dedicated to background information and motivation of the research presented in this paper. It briefs the reader on the current state of the art and known limitations in service-based cloud environments and also summarises existing research efforts in the area of managing the ever-expanding complexity of cloud application platforms. Section "Related technology: stream reasoning" introduces the reader to Stream Reasoning – a promising combination of traditional stream processing systems with Semantic Web technologies. This section also explains why these techniques are suitable for developing monitoring and analysis mechanisms. In Section "Description of the framework" we present the autonomous framework: we present a high-level conceptual architecture based on the MAPE-K reference model, then describe the prototype implementation of the framework, and finally summarise our initial experimental results.

Background and motivation

A fundamental goal of cloud computing is to achieve economies of scale by providing seemingly unlimited access to computing resources, while at the same time avoiding the sunk costs associated with acquiring dedicated systems and personnel. The core underlying architecture of the cloud is, accordingly, one of service-oriented computing (SOC): services are provided as basic building blocks from which applications can be constructed both rapidly and cheaply, without compromising on reliability or security [2]. Today's providers consequently need to host an ever-increasing number of online services, and make these available both reliably and securely to large numbers of users spread across a wide range of geographical locations. Meeting these demands has naturally shaped the way services are provided: it has long been recognised that the 'service cloud' will ultimately comprise a federated collection of resources distributed across multiple infrastructure providers [3], and cloud application platforms can be expected to play an important role in this context. Creating and maintaining the required infrastructure is, inevitably, an increasingly complex issue, and one that needs careful consideration.

As in all industries, cloud service providers face the problem of monitoring their customers' changing needs, and responding in an appropriate and timely manner. This is particularly problematic in the context of cloud services, because these are, by definition, targeted at users whose needs can be expected to change both rapidly and, at times, dramatically. Providers therefore need to monitor service usage in real time, so as to identify bottlenecks and failures that might undermine their ability to honour their customers' service-level agreements (SLAs) – and having identified 'broken' services, these need to be replaced seamlessly with new services whose behaviour is, in some contractually meaningful sense, equivalent to those being replaced. Given that services also need to be ubiquitous and available to customers using them in new and potentially unexpected ways, it is clear that cloud services will need to become increasingly autonomous and self-describing, reusable, and highly interoperable. This is particularly important at the PaaS level, given the large and increasing number of generic platform services and apps that are available, offering everything from basic calculator functionality to multi-environment distributed business processing [4].

Such cloud platforms, which not only provision customers with an operating system and run-time environment, but additionally offer a complete supporting environment to develop and deploy service-based applications, including a range of generic, reliable, composable and reusable services, are known as cloud application platforms (CAPs) [5,6] (see Figure 1). By offering integrated services in this way, CAPs further reduce the human effort and capital expenses associated with developing complex software systems. This means that software developers – CAP 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 CAP. 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. Some of the most prominent and commercially successful CAPs already provision their subscribers with tens of built-ins and third-party services. For example, Google App Engine [7] offers 38 services, and Microsoft Azure [8] provides 20 built-in services and 35 add-ons (i.e., third-party services

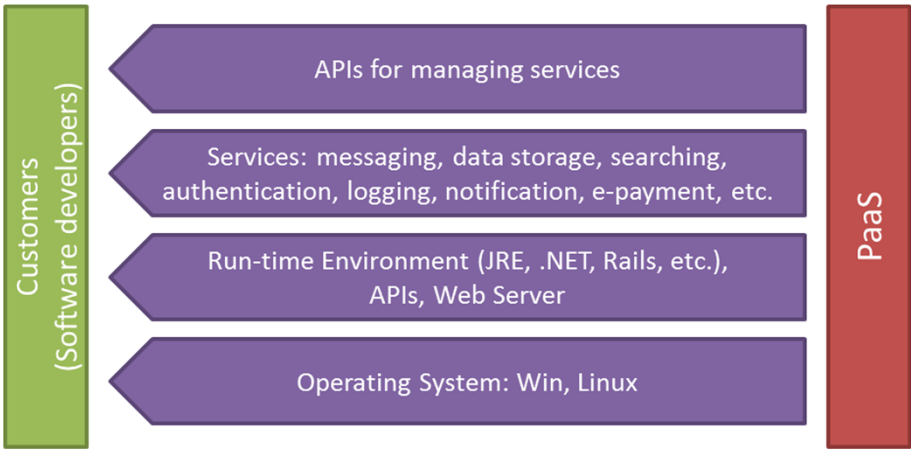

Figure 1 General structure of a Cloud Application Platform (CAP). By offering integrated services together with PaaS provision, CAPs reduce the costs associated with complex development.

registered with the platform). Appealing opportunities to significantly decrease time to market, introduced by the combination of cloud computing and SOC, has been attracting more and more attention over the past several years. Gartner forecasts that the PaaS market will grow from $900 M (in 2011) to $2.9B (in 2016) with the aPaaS (application Platform-as-a-Service) as the largest segment [9]. IDC, another leading IT market research and analysis agency, predicts that in 2014 "value will start to migrate "up the stack", from IaaS to PaaS and from generic PaaS to data-optimized PaaS" [10].

From SOC to clouds and its consequences

Given the continuing shift towards cloud computing, the complexity of next-generation service-based cloud environments is soon expected to outgrow our ability to manage them manually [11,12]. For instance, Heroku [13] already offers more than 120 different 'add-ons', including such services as data storage, searching, e-mail and SMS notification, logging and caching, and more. These can be re-used and integrated by users, generating complex interrelationships between services and user applications – indeed, add-ons are already being replicated across multiple computational instances, coupled to more than a million deployed applications [14,15].

Maintaining the ever-expanding software environment of a CAP is, consequently, a major challenge. Platform providers must be able to monitor the resulting "tangled" environment for failures and sub-optimal behaviours, while simultaneously addressing the needs of customer SLAs. They must be able to exercise control over all critical activities taking place on the platform, including the introduction of new services and applications and the modification of existing ones to maintain the platform's and deployed applications' stability and performance [16]. As we have argued above, this requires the introduction of autonomic features to the system, thereby allowing services, and the platform as a whole, to adapt their behaviours as required, following the principles of self-management, self-tuning, self-configuration, self-diagnosis, and self-healing [17].

While cloud providers arguably offer suitable adaptation mechanisms at the Infrastructure-as-a-Service (IaaS) level [18] - mainly dealing with load-balancing and elasticity - the same does not appear to be true at the PaaS level, where providers do not currently provide prompt, timely, and customisable self-management mechanisms [4] – mechanisms which would support intelligent, flexible, prompt and timely analysis of monitored values and detection of potential failures. At the PaaS level, a vast stream of data is constantly being generated and processed by a far wider range of agents than are present at the IaaS level, including a wide variety of platform components, generic and third-party services, deployed applications, and more.

For instance, WhatsApp – the world's leading instant messaging application for mobile devices [19] – is hosted on Google App Engine, utilises its XMPP-compatible chat messaging service and reports activity of 400 million monthly active users [20]. Heroku also reports several notable examples [21]: *PageLever*, an analytics platform for measuring a brand's presence on Facebook, processes 500 million Facebook API requests/month, which are then stored in a database. *Quiz Creator* saw activity peaks of over 10,000 user requests/minute. *Playtomic*, an application for run-time game analytics, claims to have around 15-20 million gamers generating over a billion events per day at the rate of 12,000 requests/second. Heroku itself hosts over one million deployed applications at a smaller scale and offers more than one hundred add-ons (20 of

which are purely concerned with data storage). As these examples demonstrate, systems as dynamic as CAPs must handle numerous rapidly-generated streams of raw data at an unpredictable rate – that is, they must be capable of performing continuous monitoring and analysis of all critical activities taking place within the platform in order to maintain the overall stability of the platform and hosted applications.

State of the art in cloud self-management

Maintaining cloud environments has been a task of paramount importance ever since the emergence of cloud computing. Such complex environments clearly dictate the need for automated monitoring and analysis of vast amounts of dynamically flowing data so as to perform, for example, resource planning and management, billing, troubleshooting, SLA and performance management, security management, etc. [18].

According to [22,23], a cloud can be logically represented in terms of seven interconnected layers: facility, network, hardware, operating system (OS), middleware, application and user. Accordingly, monitoring and analysis activities can be performed at each of these layers or in a cross-layer manner. Based on this taxonomy, Aceto et al. [18] recently surveyed 28 existing cloud monitoring tools and solutions with respect to such criteria as scalability, timeliness, autonomicity, adaptability, reliability, accuracy, resilience, extensibility, intrusiveness and others. Most of the analysed works are specifically designed to perform low-level monitoring [24] (at facility, network, hardware and OS levels) – that is, to monitor the Infrastructure-as-a-Service (IaaS) level of cloud computing. Performing monitoring activities at this level primarily enables cloud providers to adapt to varying volumes and types of user requests by allocating the incoming workload across computational instances (i.e., load balancing), or by reserving and releasing computational resources upon demand (i.e., elasticity) [25,26].

However, more sophisticated adaptation scenarios at higher levels (middleware and application), such as modifying the actual structure and/or behaviour of a deployed application at run-time, are much more difficult to automate, and are currently beyond the capabilities of common CAPs. Unfortunately, at the moment there seem to be no self-management mechanisms of such a kind at the Platform-as-a-Service (PaaS) level. Even though there are several approaches which perform monitoring at the middleware level, the values they collect are primarily used to perform adaptations at theIaaS, rather than PaaS, level – for example, instead of replacing a "slow" service with an equivalent (but faster) alternative (PaaS-level adaptation), additional computational resources are provisioned to the given service (IaaS-level adaptation).

An alternative approach to PaaS-level adaptations performed by CAP providers is to require deployed applications to implement their own built-in adaptation functionality. As with IaaS solutions, this means that platform providers do not offer 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 substituting one service for another). Instead, this task has been shifted to the Software-as-a-Service (SaaS) level – that is, it has been left to software developers, the target customers of the PaaS offerings, to implement self-adaptation logic within their applications.

Given these considerations, we believe that self-adaptation capabilities at the PaaS level itself, and in CAPs in particular, are as yet immature and not well theorised. It is our belief that self-management at the PaaS level is equally important, and that development of self-adaptation mechanisms at this level is essential in order to prevent cloud platforms from dissolving into "tangled" and unreliable environments. Our goal in this paper is to present and justify one possible strategy for addressing this gap.

Related technology: stream reasoning

Since the early 2000s, when data volumes started exploding, the challenge of data analytics has grown considerably. Nowadays, the problem is not just about giant data volumes ("Big Data") – it is also about an extreme diversity of data types, delivered at various speeds and frequencies [27]. In the modern world, heterogeneous data streams are to be found everywhere – sensors networks, social media sites, digital pictures and videos, purchase transaction records, and mobile phone GPS signals, to name a few [28] – and on-the-fly processing of newly generated data has become an increasingly difficult challenge. Two important aspects of traditional database management systems make them unsuitable for processing continuously streamed data from geographically distributed sources at unpredictable rates, so as to obtain timely responses to complex queries [29], namely: (i) data is (persistently) stored and indexed before it can be processed, and (ii) data is processed only when explicitly queried by users, i.e. asynchronously with respect to its arrival. In contrast, streamed data cannot sensibly be stored for any length of time if it is to be used for real-time adaptation; and we cannot rely on users issuing one-off queries. Rather, we need some way for adaptation to be triggered automatically, as and when problems arise.

To cope with the unbounded nature of streams and temporal constraints, so-called continuous query languages [30] have been developed to extend conventional SQL semantics with the notion of windows. This approach restricts querying to a specific window of concern which consists of a subset of statements recently observed on

the stream, while older information is (usually) ignored, thereby allowing traditional relational operators to be applied [31].

The concepts of unbounded data streams and windows are visualised in Figure 2. The small 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 are pushed out of the specified window, i.e. they are deemed irrelevant and may be discarded (unless there is a need for storing historical data for later analysis).

Stream Reasoning goes one step further by enhancing continuous queries with run-time reasoning support – that is, with capabilities to infer additional, implicit knowledge based on already given, explicit facts. The concept was introduced by Barbieri et al. [32], who defined it as "reasoning in real time on huge and possibly noisy data streams, to support a large number of concurrent decision processes". In the Big Data paradigm, for example, where data streams are becoming increasingly pervasive, the combination of stream processing techniques with dynamically generated data, distributed across the Web, requires new ways of coping with the typical openness and heterogeneity of the Web environment – in this context, Semantic Web technologies facilitate data integration in open environments, and thus help to overcome these problems by using uniform machine-readable descriptions to resolve heterogeneities across multiple data streams [33]. The primary segment of Big Data processing, where Stream Reasoning is being adopted is the Semantic Sensor Web [34] – "an attempt to enable more expressive representation and formal analysis of avalanches of sensor values in such domains as traffic surveillance, environmental monitoring, house automation and tracking systems, by encoding sensor observation data with Semantic Web languages" [35]. Other problem domains, where Stream Reasoning techniques are expected to be effective, include, e.g., analysis of social media streams, understanding users' behaviour based on their click streams, and analysis of trends in medical records to predict spread of a disease over the world. [36].

As Semantic Web technologies are mainly based on Description Logics, their application to data stream processing also offers new opportunities to perform reasoning tasks over continuously and rapidly changing flows of information. In particular, Stream Reasoning utilises and benefits from the following Semantic Web technologies:

- Resource Description Framework (RDF), as a uniform format for representing streamed heterogeneous data as a collection of (subject, predicate, object) triples using a vocabulary defined in an OWL ontology;
- OWL ontologies and SWRL rules, as a source of static background knowledge. OWL ontologies may also act as a vocabulary of terms for defining RDF triples;
- SPARQL-based continuous query languages, as a way of querying RDF streams and performing reasoning tasks by combining them with the static background knowledge.

As a result, several prominent Stream Reasoning approaches have emerged, including, e.g., C-SPARQL [37], CQELS [38], ETALIS [39], and SPARQLstream [30]. These systems aim at preserving the core value of data stream processing, i.e. processing streamed data in a timely fashion, while providing a number of additional features [33]:

- Support for advanced reasoning: depending on the extent to which Stream Reasoning systems support reasoning, it is possible not only to detect patterns of events (as Complex Event Processing already does [29]), but in addition to perform more sophisticated

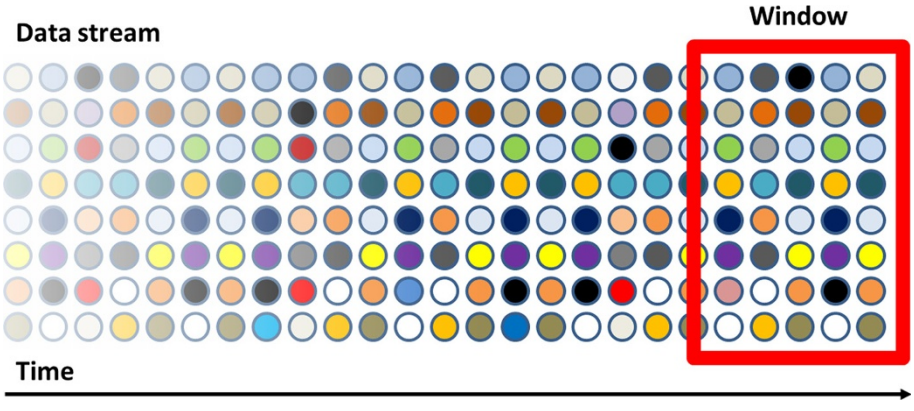

Figure 2 Querying of streamed data. Continuous query languages address the problem of querying an unbounded data stream by focussing on a well-defined window of interest.

and intelligent detection of failures by inferring implicit knowledge based on pre-defined facts and rules (i.e., static background knowledge).
- Integration of static background knowledge with streamed data: it is possible to match data stream values against a static background knowledge base (usually represented as an ontology), containing various facts and rules. This separation of concerns allows for seamless and transparent modification of the analysis rules constituting the static knowledge base.
- Support for expressive queries and complex schemas: ontologies also serve as a common vocabulary for defining complex queries. This means that the classes and properties constituting an ontology provide "building blocks" and may be used for defining queries of any required expressivity.
- Support for logical, data and temporal operators: to cope with the unlimited nature of data streams, Stream Reasoning systems extend conventional SQL-based logical and data operators with temporal operators. This allows us to limit an unbounded stream to a specific window, and also to detect events following one after another chronologically.
- Support for time and tuple windows: windows may be specified either by time-frame, or else by the number of entries to be retained, regardless of arrival time. Taken together, these features facilitate evaluation of expressive queries over streamed data and, as a result, have the potential to allow us to benefit from increased analysis capabilities when processing data streams, such as monitored values within CAPs. However, no solution is ever perfect, and Stream Reasoning at its current state is not an exception. Accordingly, in order to realise its potential in the context of analysing large data streams of CAPs, we need to address following shortcomings [35]:
- Need for unified data representation format: before formal reasoning can be applied, heterogeneous values have to be represented in a common format – RDF. Although this process can be seen as a way of tackling, e.g., the "variety" aspect of Big Data [40], it requires establishing mappings between source data formats and their RDF representations, which has to be performed manually.
- No standards yet: the lack of common standards resulted in several independent and hardly interoperable approaches.
- Immature reasoning support: as opposed to static SPARQL reasoning capabilities, querying over dynamic data streams with the reasoning support is not fully implemented yet, and the conventional SPARQL 2.0 specification is not supported by any of the existing Stream Reasoning approaches.
- Low performance: the Stream Reasoning research area is still in its infancy, and suffers from low performance. Since expressivity of a query language is known to be inversely related to its performance [33], evaluation of rich and complex queries always bears the penalty of performance, which is particularly critical when performing data analytics on very large data streams.
- Low scalability: one of the main shortcomings of formal reasoning, both static and stream, is that it is not linearly scalable [41]. This means that the larger the knowledge base over which reasoning is performed, the slower this process is. In the context of analysing large data sets within CAPs, this shortcoming becomes a major concern and cannot be neglected.

In summary, Stream Reasoning is not a "silver bullet" – its shortcomings, unless properly addressed, may outweigh its positive aspects and seriously hinder the implementation of the autonomous framework.

Description of the framework

In this section we explain the underlying organisation of the autonomous framework, starting from a high-level description of the architecture and then going into implementation details. Through our experiments with Heroku, we discovered that simply deploying the autonomous framework to a cloud is not enough. Shortcomings associated with Stream Reasoning required us to address unexpected performance and scalability issues relating to our approach, as will be further discussed in Subsection "Evaluation and future work" below.

Conceptual architecture

We will describe our approach by sketching out a high-level architecture of the framework, taking the established MAPE-K framework [17] as our underlying model for self-adaptation (see Figure 3). In order to support both self-awareness and context-awareness of the managed elements, we need to employ some kind of architectural model describing the adaptation-relevant aspects of the cloud environment (e.g., platform components, available resources, connections between them, etc.) and the managed elements (e.g., entry-points for monitoring and execution). We therefore used OWL ontologies to represent the self-reflective knowledge of the system. Such an architectural model also serves as a common vocabulary of terms shared across the whole managed system, and corresponds to the Knowledge component of the MAPE-K model. Moreover, our ontological classes and properties, as explained below, also serve as "building blocks" for creating RDF streams, SPARQL queries and SWRL rules.

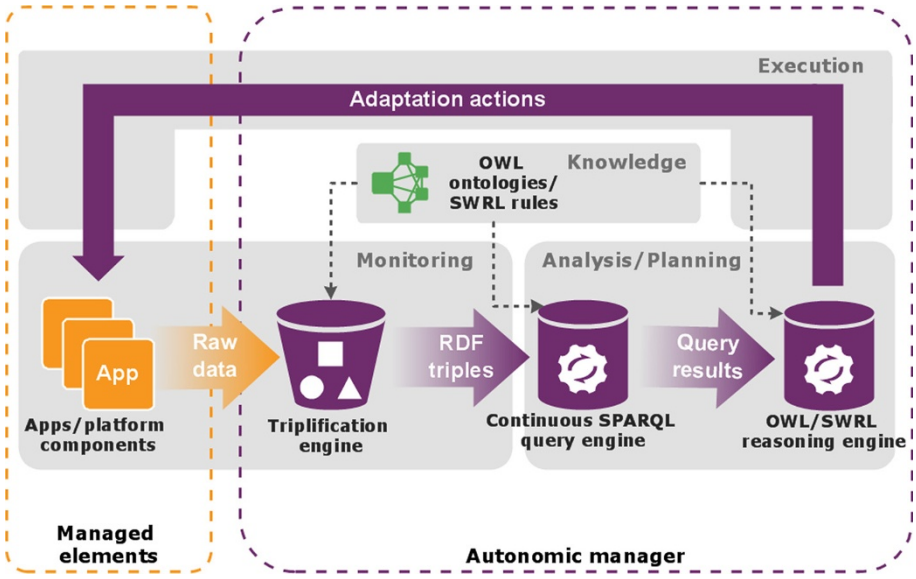

Figure 3 Prototype framework. Our framework architecture implements the established MAPE-K control model for autonomic computing.

Within the framework, raw data generated by sensors passes through three main processing steps:

- The triplification engine is a software component responsible for consuming and "homogenising" the representation of incoming raw observation values. The use of time-stamped RDF triples, incorporating OWL-based subjects, predicates and objects, promotes human-readability while at the same time allowing us to exploit the extensive capabilities of SPARQL query languages.
- The continuous SPARQL query engine is a software component which supports situation assessment by taking as input the continuous RDF data streams generated by the triplification engine and evaluating them against pre-registered continuous SPARQL queries. By registering appropriate SPARQL query against a data stream, we are able to detect critical situations – for example, service failures, high response time from services, overloaded message queues and network request time outs – with minimal delay: the continuous SPARQL engine will trigger as soon as RDF triples in the stream match the WHERE clause of any registered query. Using SPARQL and RDF triples in this way also makes it possible to benefit from inference capabilities – in addition to querying data and detecting complex event patterns, we are able to perform run-time analysis by reasoning over RDF triples [35]. Employing existing RDF streaming engines with "on-the-fly" analysis of constantly flowing observations from hundreds of sensors is expected to help us achieve near real-time behaviour [36] of the adaptation framework (as opposed to "static" approaches where monitored data is first stored on the hard drive before being analysed) – a key requirement when developing an adaptation mechanism.
- The OWL/SWRL reasoning engine is the software component responsible for generating a final diagnosis and an appropriate adaptation plan whenever a critical condition is detected, a process which typically requires rather complex reasoning over the possible roots of a problem, and the identification of multiple potential adaptation strategies. We address this challenge (at least partially) using OWL ontologies and SWRL rules, since these provide sufficient expressivity to define adaptation policies [42], while at the same time avoiding the potentially error-prone and intensive task of implementing our own analysis engines from scratch. Instead, we apply the built-in reasoning capabilities of OWL and SWRL, so that the routine of reasoning over (i.e., analysing) a set of situations and adaptation alternatives is achieved using an existing, tested, and highly optimised mechanism. This also enhances opportunities for reuse, automation and reliability [12].

Prototype implementation

As a first step towards a proof of concept, a prototype solution implementing the conceptual architecture has been developed. As a test bed for our experiments we have chosen Heroku – a well-established and trustable

cloud PaaS offering with sufficient levels of support and documentation for our purposes. As outlined in Section "Background and motivation", Heroku qualifies as a cloud application platform, and it offers a range of add-on services to experiment with. The main criteria used when choosing services for our experiments were: (i) pervasive use by deployed applications to make sure that the service is widely used and thus emits enough data to be monitored and analysed, (ii) easy and straightforward provisioning of the service, and (iii) presence of clear and simple metrics for monitoring. Accordingly, from the more than 100 Heroku add-ons available, we chose for our experiments an implementation of the RabbitMQ messaging queue service called CloudAMQP [43] – a widely adopted solution for decoupled communication between various components of cloud-based distributed applications. We then developed the following simple use-case scenario, both to test the viability of our general approach, and to identify directions for further work and experimentation.

Let us consider a typical cloud-based scenario, in which worker applications responsible for background processing of various tasks are decoupled from the main application by means of a messaging queue service. At some point, the job queue gets overloaded and workers spend too much time processing their tasks. This may happen, for example, on a cheaper subscription plan when the rate of incoming messages is faster than the queue service can write to disk, or the volume of incoming messages exceeds the available memory [44]. In order to satisfy SLAs we wish to detect such situations in a timely manner and, where possible, launch additional worker instances to offload the job queue. In investigating this scenario our main focus has been on the monitoring and analysis steps of the MAPE-K model, whereas the planning and execution steps have been left aside (this example is intended only to demonstrate the viability of our approach, and is correspondingly simplified).

Figure 4 shows the architecture of the prototype and illustrates the steps constituting the use-case scenario. The client application sends tasks to the RabbitMQ job queue, which are then picked up and processed by an available worker instance on a first-come first-served basis. Once a job task is processed, the worker acknowledges the queue of accomplishing the task.

Accordingly, in order to support proper functioning of this simple application system, we are interested in monitoring the following metrics associated with the messaging queue:

(i) Size of the message queue (i.e., current number of messages in the queue).
(ii) Message queuing time (i.e., difference between the time when a message is published to the queue and the time when it is consumed by a worker).

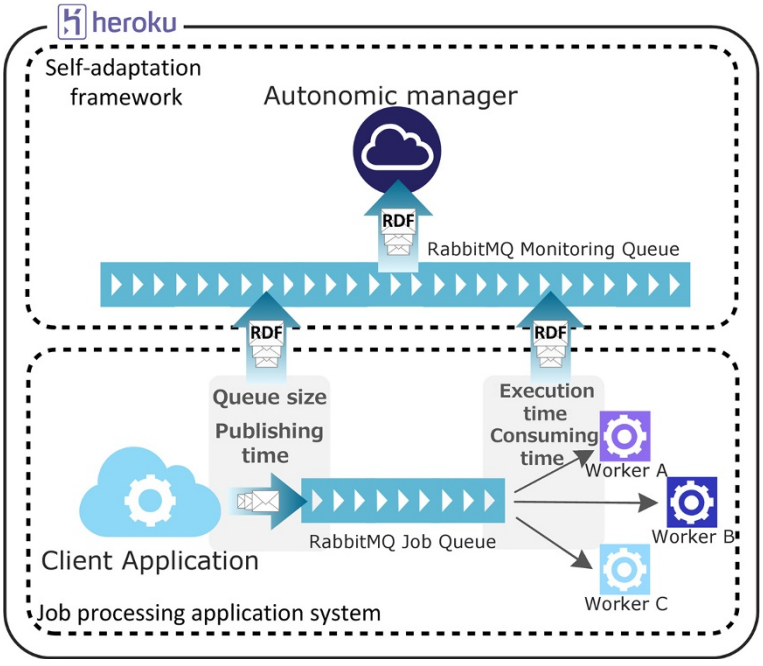

Figure 4 Prototype architecture. The prototype architecture used in our proof-of-concept use-case scenario involves real-time monitoring of service performance by an Autonomic Manager. The Autonomic Manager responds to critical situations at the PaaS level by identifying when 'slow' services should be replaced by faster alternatives.

(iii) Workers' execution time (i.e., time between the moment when a worker picks up a message from the queue and the moment when it acknowledges the queue of accomplishing the task).

We chose these particular metrics because they are easily monitored within Heroku (hence suitable for test purposes), and when violated they unambiguously indicate a critical situation. Simultaneous violation of threshold values of these three metrics represents a critical situation where the current number of worker instances is not able to cope with the workload, so that additional instances have to be launched to prevent the whole system from crashing.

To implement this simple use-case, we used Java as a programming language, Eclipse IDE (with the Heroku plug-in) for coding and testing, and Protégé IDE [45] for developing the OWL ontology and SWRL rules. We also used OWL API [46] to create, manipulate and reason over the OWL ontology and SWRL rules programmatically, and the C-SPARQL library [47] to create and query RDF streams.

The main components we implemented were:

- The RabbitMQ job queue – used to decouple the client application from workers, and for transferring tasks. The CloudAMQP implementation of RabbitMQ, offered by Heroku, is easy to use and configure within Java applications.
- Client – a GUI application responsible for sending jobs to the corresponding queue. From the GUI it is possible to specify the number of parallel threads sending tasks to the queue.
- Workers – computational instances responsible for picking tasks from the queue, executing them and notifying the queue when the task is accomplished.
- Sensors – pieces of programming code responsible for measuring: (i) size of the message queue – every time the client sends a new task to the queue, it receives back the current number of awaiting messages; (ii) time when a new task is published and time when it is consumed by a first available worker; (iii) workers' execution times – workers calculate their own execution time by subtracting the time when a job was picked up from the queue from the time when it was processed. The measured values are then transformed into RDF triples using terms from the OWL vocabulary, and sent to the RabbitMQ monitoring queue. The following three samples demonstrate how data, generated by sensors as described in the above-mentioned cases, is represented in the RDF format (where ex is shorthand notation for a purpose-built ontology containing corresponding classes and properties):

```
(i)   ex:queue13 ex:hasQueueSize ex:QS81
      ex:QS81    ex:hasValue "15500"^^xsd:int
(ii)  ex:task782 ex:isPublishedAt ex:time231
      ex:time231 ex:hasValue "11:12:13.555"^^xsd:time
      ex:task782 ex:isConsumedAt ex:time298
      ex:time298 ex:hasValue "11:12:19.213"^^xsd:time
(iii) ex:worker7 ex:hasExecutionTime ex:ET98
      ex:ET98    ex:hasValue "11.54"^^xsd:float
```

The main components constituting the monitoring framework are:

- The RabbitMQ monitoring queue – used to collect monitored values of the job queue workload and workers' response times.
- Autonomic Manager – this is the core component of the framework responsible for collecting and analysing monitored values, detecting/predicting critical conditions, and generating corresponding adaptation actions. By registering appropriate C-SPARQL queries against the monitoring queue, the Autonomic Manager is notified as soon as the RDF triples in the stream satisfy the WHERE clause of the query. Let us now consider the following queries for each of the monitored metrics:

```
(i) REGISTER QUERY QueueSizeQuery
    AS PREFIX ex:<http://seerc.org/ontology.owl#>
    SELECT ?queue ?qs ?v
    FROM STREAM <http://seerc.org/stream>
       [RANGE 1m STEP 1s]
    WHERE {?queue ex:hasQueueSize ?qs .
       ?qs ex:hasValue ?v . FILTER (?v >= 10000)}
```

This query is triggered whenever the number of awaiting messages exceeds 10000.

```
(ii) REGISTER QUERY QueuingTimeQuery
     AS PREFIX ex:<http://seerc.org/ontology.owl#>
     SELECT ?q ?qs ?v
     FROM STREAM <http://seerc.org/stream>
        [RANGE 1m STEP 1s]
     WHERE {?task ex:isPublishedAt ?t1 .
        ?t1 ex:hasValue ?v1 .
        ?task ex:isConsumedAt ?t2 .
        ?t2 ex:hasValue ?v2 .
         FILTER ((?v2 - ?v1) > 10000)}
```

This query is triggered whenever the difference between the time when a message is published and the time when it is consumed exceeds 10 seconds.

```
(iii) REGISTER QUERY ExecutionTimeQuery
      AS PREFIX ex: <http://seerc.org/ontology.owl#>
      SELECT ?worker ?t ?v
      FROM STREAM <http://seerc.org/stream>
         [RANGE 1m STEP 1s]
      WHERE {?worker ex:hasExecutiontime ?t .
        ?t ex:hasValue ?v . FILTER (?v >= 5000)}
```

This query is triggered whenever the execution time of a worker exceeds 5 seconds.

Querying over the RDF stream allows the autonomic manager to detect if there are too many messages in the queue, and to identify execution time violations. Once queries are triggered, fetched values are added to the OWL ontology in order for traditional static reasoning to be applied. The Autonomic Manager is able to deduce that the overloaded queue and workers represent a critical situation for which a possible adaptation strategy would be the launching of additional worker instances. These activities are performed by reasoning over the OWL ontology and SWRL rules, which are declaratively defined by platform administrators at design-time with respect to given SLAs, and can be modified at run-time if needed. The following is illustrates a typical SWRL rule, which allows the Autonomic Manager, based on the violated values received at the querying stage, to deduce that the observed critical situation requires some adaptation actions – that is, that additional worker instances have to be launched:

```
Worker(?w), Queue(?q), isSubscribedTo(?w, ?q),
   hasQueueSize(?q, ?qs), hasValue(?qs, ?v1),
   greaterThan(?v1, 10000), Task(?t),
   hasQueuingTime(?t, ?qt), hasValue(?qt, ?v2),
   greaterThan(?v2, 10000),
   hasExecutionTime(?t, ?et), hasValue(?et, ?v3),
   greaterThan(?v3, 5000)
    -> needsAdditionalInstances(?w, true)
```

We have run initial experiments on Heroku's Cedar stack using a free account – each computational instance (or dyno in Heroku terminology) has 512 MB of RAM and 1GB of swap space (1.5 GB RAM in total), and 4 CPU cores (Intel Xeon X5550 2.67 GHz). Single instances of the autonomic manager and the client application and three worker instances were deployed on separate dynos. To simulate the critical workload on the queue we: (i) completely turned workers off to let messages accumulate to reach critical level (the threshold level we specified in a corresponding C-SPARQL queue was set to 10000 messages); and (ii) made workers "sleep" for 1000 ms every time they picked a task from the queue (the threshold level of message queuing time was set to 10000 ms, and the threshold level of workers' execution times was set to 5000 ms). These initial experiments show that we are able to detect all critical conditions within 1 second – this is the minimum time frame between two consecutive evaluations of registered queries against the data stream which is allowed by the current implementation of the C-SPARQL engine.

Evaluation and future work

Unfortunately, as the number of incoming RDF triples increases, the performance of the framework decreases. As explained in Section "Related technology: stream reasoning", Stream Reasoning on its own currently suffers from performance and scalability issues. Existing experiments suggest that with the increase of RDF data sets from 10K to 1M triples, the average execution time of C-SPARQL queries increases at least 50 times [38]. Such performance drops make our framework potentially incapable of monitoring and analysing large data sets within CAPs and need to be addressed, especially if at some later stage we wish to expand the technique to handle Big Data scenarios.

A possible solution to this problem, to be investigated in the next stage of our work, is to parallelise reasoning tasks across several instances of the Autonomic Manager [48] by fragmenting incoming data streams into sub-streams, so that each instance only deals with a subset of the incoming values. 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. Therefore, careful design of the fragmentation logic is crucial in order to confirm that no valuable data is misplaced or lost.

In order to address this challenge, there already exist several technologies, both commercial (e.g., Oracle Fast Data solutions [49] and IBM InfoSphere Streams [50]) and open-source (e.g. Apache S4 [51] or Storm [52]). These solutions provide infrastructure and tooling in order to handle massive data streams, and as the next step we will integrate our autonomous framework with one of these data stream solutions. We anticipate that this will allow us to address two of the five Stream Reasoning challenges identified in Section "Related technology: stream reasoning", namely, scalability and performance.

We also want to emphasise that we expect our main contribution to be in the area of prompt, dynamic and intelligent analysis of the monitored values (which is quite difficult to benchmark), rather than in terms of performance. We also anticipate that further on-going developments in Stream Reasoning will see the resolution of two further shortcomings – the immature reasoning support and the lack of standards. The requirement to homogenise data and represent it in RDF format is expected to be less problematic. There already exist tools for converting data stored in relational databases into RDF, using special mapping languages (e.g., R2RML [53]), and analogous tools can be envisaged for RDF stream generation.

Our prototype case study suggests that once we have implemented the whole MAPE-K chain for a small number of key parameters (e.g., the number of messages in the queue, message queuing time, and the execution times from workers), the introduction of additional monitoring parameters becomes a trivial task, and does not

necessarily bring scientific contribution. Rather than create a comprehensive adaptation framework which would monitor and analyse all possible metrics of a cloud platform, our future research activities will therefore focus on further enriching the background knowledge base (i.e. OWL/SWRL policies) to see what kind of knowledge can be inferred from a limited number of observed parameters to support the analysis and diagnosis of potential failures.

We also plan to experiment with other continuous query engines, such as CQELS and SPARQLstream, and compare them in terms of the analytical support they can offer. At the moment, on-the-fly reasoning support of continuous SPARQL query languages is quite limited (at least compared to traditional static SPARQL) [35], and depends on the supported entailment regimes of particular query languages. Research in the direction of bridging the gap between static and dynamic reasoning support in SPARQL queries is continuing, and we can also reasonably hope for truly run-time reasoning to appear in the relatively near future.

Conclusion

In this paper we have presented a novel approach to enhancing cloud platforms with self-managing capabilities. It utilises the Semantic Web technology stack for annotating observation values with semantic descriptions, and techniques from Stream Reasoning for performing run-time analysis and problem diagnosis within cloud application platforms. We have also introduced a conceptual architecture which follows the MAPE-K reference model to implement closed adaptation loops, and a prototype framework developed in Java and deployed on Heroku. Initial experiments demonstrate the viability of the proposed approach, both in terms of performance and in terms of the analysis capabilities of the autonomous framework. More specifically, the framework is able not only to monitor values, but also to detect and diagnose critical situations, and to propose a simple adaptation strategy within 1 second. Our results are, however, based on a relatively small-scale case study, and we have identified further challenges associated with Stream reasoning that will need to be overcome for the approach to become adopted in practice. Even so, we believe that the application of Stream Reasoning and Semantic Web – two areas where intelligence lies at the very core – to Autonomic Clouds is a promising direction.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

The research presented in this paper is part of the Ph.D. dissertation of the first author under the supervision of the second and the third authors. All authors equally contributed to the paper. All authors read and approved the final manuscript.

Acknowledgements

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.

Author details

[1]South-East European Research Centre, International Faculty of the University of Sheffield, City College, 24 Proxenou Koromila Street, 54646 Thessaloniki, Greece. [2]Department of Computer Science, University of Sheffield, Regent Court, 211 Portobello Street, S1 4DP Sheffield, UK.

Received: 9 January 2014 Accepted: 30 June 2014

References

1. Dautov R, Paraskakis I (2013) A vision for monitoring cloud application platforms as sensor networks. In: Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference. CAC '13. ACM, New York, pp 25–1258
2. Papazoglou MP, Traverso P, Dustdar S, Leymann F (2008) Service-oriented computing: a research roadmap. Int J Coop Inf Syst 17(2):223–255
3. Clayman S, Galis A, Chapman C, Toffetti G, Rodero-Merino L, Vaquero LM, Nagin K, Rochwerger B (2010) Monitoring service clouds in the future internet. In: Tselentis G, et al. (eds) Towards the future internet – Emerging trends from European research. IOS Press, Amsterdam, pp 105–114
4. Wei Y, Blake MB (2010) Service-oriented computing and cloud computing: Challenges and opportunities. Internet Comput IEEE 14(6):72–75
5. Kourtesis D, Bratanis K, Bibikas D, Paraskakis I (2012) Software co-development in the era of cloud application platforms and Ecosystems: The case of CAST. In: Camarinha-Matos LM, Xu L, Afsarmanesh H (eds) Collaborative networks in the internet of services, vol. 380. Springer, Berlin Heidelberg, pp 196–204
6. Ried S, Rymer JR (2011) The Forrester Wave™: Platform-As-A-Service For Vendor Strategy Professionals. Q2 2011. Technical report, Forrester Research, Cambridge, MA, USA
7. Google App Engine. http://appengine.google.com/
8. Windows Azure. http://www.windowsazure.com/
9. Gartner Says Worldwide Platform as a Service Revenue Is on Pace to Reach $1.2 Billion. http://www.gartner.com/newsroom/id/2242415
10. IDC Predicts 2014 Will Be a Year of Escalation, Consolidation, and Innovation as the Transition to IT's "Third Platform" Accelerates. http://www.idc.com/getdoc.jsp?containerId=prUS24472713
11. Brazier FM, Kephart JO, Van Dyke Parunak H, Huhns MN (2009) Agents and service-oriented computing for autonomic computing: a research agenda. Internet Comput IEEE 13(3):82–87
12. Dautov R, Kourtesis D, Paraskakis I, Stannett M (2013) Addressing self-management in cloud platforms: a semantic sensor web approach. In: Proceedings of the 2013 international workshop on hot topics in cloud services. ACM, New York, pp 11–18
13. Heroku. http://www.heroku.com/
14. Harris D Heroku Boss: 1.5M Apps, Many Not in Ruby. http://gigaom.com/2012/05/04/heroku-boss-1-5m-apps-many-not-in-ruby/
15. Heroku – CrunchBase Profile. http://www.crunchbase.com/company/heroku
16. Kourtesis D (2011) Towards an ontology-driven governance framework for cloud application platforms. Technical report. South-East European Research Centre (SEERC), Thessaloniki, Greece
17. Kephart JO, Chess DM (2003) The vision of autonomic computing. Computer 36(1):41–50
18. Aceto G, Botta A, de Donato W, Pescapè A (2013) Cloud monitoring: a survey. Comput Netw 57(9):2093–2115
19. WhatsApp Leads The Global Smartphone Messenger Wars With 44 Percent Market Share. http://www.1mtb.com/whatsapp-leads-the-global-mobile-messenger-wars-with-44-pc-market-share/
20. 400 Million Stories. http://blog.whatsappcom/index.php/2013/12/400-million-stories/
21. Heroku – Success. http://success.heroku.com/
22. Spring J (2011) Monitoring cloud computing by layer, part 1. Secur. Priv. IEEE 9(2):66–68
23. Spring J (2011) Monitoring cloud computing by layer, part 2. Secur Priv IEEE 9(3):52–55

24. Caron E, Desprez F, Rodero-Merino L, Muresan A (2012) Auto-scaling, load balancing and monitoring in commercial and open-source clouds. In: Benatallah B (ed) Cloud computing: methodology, systems, and applications. CRC Press, Boca Raton, pp 301–323
25. Armbrust M, Fox A, Joseph A, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2009) Above the clouds: a berkeley view of cloud computing, EECS-2009-28. Technical report, University of California Berkeley, Berkeley, CA, USA
26. Natis YV, Knipp E, Valdes R, Cearley DW, Sholler D (2009) Who's who in application platforms for cloud computing: the cloud specialists. Technical report, Gartner Research, Stamford, CT, USA
27. Russom P (2011) Big data analytics. TDWI Best Practices Report, Renton, WA, USA
28. Botts M, Percivall G, Reed C, Davidson J (2008) OGC® sensor web enablement: overview and high level architecture. In: Nittel S, Labrinidis A, Stefanidis A (eds) GeoSensor Networks. Lecture Notes in Computer Science, vol. 4540. Springer, Berlin Heidelberg, pp 175–190
29. Cugola G, Margara A (2012) Processing flows of information: from data stream to complex event processing. ACM Comput Surv (CSUR) 44(3):15
30. Calbimonte J-P, Jeung H, Corcho O, Aberer K (2012) Enabling query technologies for the semantic sensor web. Int J Semantic Web Inform Syst (IJSWIS) 8(1):43–63
31. Barbieri D, Braga D, Ceri S, Della Valle E, Grossniklaus M (2010) Stream reasoning: where we got so far. In: Proceedings of the 4th workshop on new forms of reasoning for the semantic web: scalable & dynamic. Springer, Berlin Heidelberg, pp 1–7
32. Della Valle E, Ceri S, Barbieri DF, Braga D, Campi A (2009) A first step towards stream reasoning. In: Future Internet–FIS 2008. Springer, Berlin Heidelberg, pp 72–81
33. Lanzanasto N, Komazec S, Toma I (2012) Reasoning over real time data streams. http://www.envision-project.eu/wp-content/uploads/2012/11/D4.8-1.0.pdf
34. Sheth A, Henson C, Sahoo SS (2008) Semantic sensor web. Internet Comput IEEE 12(4):78–83
35. Dautov R, Stannett M, Paraskakis I (2013) 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. Thessaloniki, Greece
36. Valle ED, Ceri S, van Harmelen F, Fensel D (2009) It's a streaming world! Reasoning upon rapidly changing information. Intell Syst IEEE 24(6):83–89
37. Barbieri DF, Braga D, Ceri S, Della Valle E, Grossniklaus M (2009) C-SPARQL: SPARQL for continuous querying. In: Proceedings of the 18th international conference on World Wide Web. ACM, New York, pp 1061–1062
38. Le-Phuoc D, Dao-Tran M, Parreira JX, Hauswirth M (2011) A native and adaptive approach for unified processing of linked streams and linked data. In: The semantic Web–ISWC 2011. Springer, Berlin Heidelberg, pp 370–388
39. Anicic D (2011) Event Processing and Stream Reasoning with ETALIS. PhD thesis, Karlsruher Institut für Technologie (KIT), Karlsruhe, Germany
40. IBM: Four Vs of Big Data. http://www.ibmbigdatahub.com/infographic/four-vs-big-data
41. Baader F (2003) The description logic handbook: theory, implementation, and applications. Cambridge University Press, Cambridge
42. Hitzler P, Krotzsch M, Rudolph S (2011) Foundations of semantic web technologies. Chapman & Hall/CRC, Boca Raton
43. CloudAMQP – RabbitMQ as a Service. https://addons.heroku.com/cloudamqp
44. Message Throughput in RabbitMQ Bigwig. http://bigwig.io/docs/message_throughput/
45. The Protégé Ontology Editor and Knowledge Acquisition System. http://protege.stanford.edu/
46. OWL API. http://owlapi.sourceforge.net/
47. Continuous SPARQL (C-SPARQL) Ready To Go Pack. http://streamreasoning.org/download/csparqlreadytogopack
48. Urbani J (2010) Scalable and parallel reasoning in the Semantic Web. In: The semantic web: research and applications. Springer, Berlin Heidelberg, pp 488–492
49. Oracle Fast Data Solutions. http://www.oracle.com/us/solutions/fastdata/index.html
50. IBM - InfoSphere Streams. http://www-03.ibm.com/software/products/en/infosphere-streams
51. Apache: S4: Distributed Stream Processing System. http://incubator.apache.org/s4/
52. Storm – Distributed and Fault-tolerant Realtime Computation. http://storm-project.net/
53. R2RML: RDB to RDF Mapping Language (W3C Recommendation 27 September 2012). http://www.w3.org/TR/r2rml/

doi:10.1186/s13677-014-0013-5
Cite this article as: Dautov *et al.*: **Utilising stream reasoning techniques to underpin an autonomous framework for cloud application platforms.** *Journal of Cloud Computing: Advances, Systems and Applications* 2014 **3**:13.