PERFORMANCE MANAGEMENT AND MONITORING

MARK SHTERN\textsuperscript{1}, BRADLEY SIMMONS\textsuperscript{1}, MICHAEL SMIT\textsuperscript{2}, HONGBIN LU\textsuperscript{1} AND MARIN LITOIU\textsuperscript{1}

\textsuperscript{1}York University, Toronto, Ontario
\textsuperscript{2}Dalhousie University, Halifax, Nova Scotia

1.1 Introduction

Organizations are transitioning from private data centers to infrastructure-as-a-service (IaaS)-style resource management where resources are acquired on-demand from a large pool, managed internally (a private cloud) or by a third party supplier (a public cloud). Interest is growing in creating a single computational fabric across a set of cloud providers, a \textit{multi-cloud} \cite{1, 2, 3, 4}. Multi-clouds are a natural evolution of cloud computing; also called the intercloud \cite{5, 6}, or clouds-of-clouds, in which multiple cloud systems (typically IaaS) are composed together to add value to users. For example, a private and public cloud can be combined to address data privacy concerns while still enjoying some public cloud benefits (i.e., \textit{hybrid clouds}, or public/private cloud overlays \cite{7}). Multiple public clouds can be federated to improve availability \cite{1}, reduce lock-in, and optimize costs \cite{8} beyond what can be achieved with a single cloud provider.

As this transition to multi-cloud progresses there are several critical differences that affect application management: i) Every application is sandboxed from every other application; ii) For an individual application the potential for resource contention is effectively zero, and iii) Resources can be acquired on-demand according to a pay-as-you-go pricing model. These differences permit applications to be more
easily managed on a per-application basis, rather than managing the entire IT infras- 
structure of an organization as a whole. This results in a shift of responsibility 
from established practices. A set of cloud providers (or, for private clouds, the 
IT operations teams) manages the physical infrastructure and provides virtualized 
containers (for IaaS, virtual machines) to clients who wish to deploy applications. 
The client assumes responsibility for both the functional and non-functional qual-
ity of a deployed application; increasingly, the client is the development team, a 
scenario referred to as *devops* and/or *noops*. Devops relies on automation for 
cost-efficient management of software systems. We provide more details about this 
transition in Section 1.2.

These changes in operational context (i.e., private datacenter versus multi-cloud) 
motivate an evolution in the approach to management of applications. If develop-
ers are expected to manage non-functional aspects of their applications, there is 
value in supporting best-practices with regard to the design and implementation 
of management logic and infrastructural support, while simultaneously incorporat-
ing established management best-practices into the overall approach. Additionally, 
the developer should be shielded from the complexity of acquiring and releasing 
resources in the context of the multi-cloud. Finally, they should be able to harness 
their own domain-specific languages (DSLs) and intimate knowledge of the applica-
tion in support of management objectives instead of being prescribed a particular 
approach.

We introduce the X-Cloud Application Management Platform (XCAMP), a 
platform to enable whomever has assumed responsibility for automating manage-
ment – application developers, researchers, operations teams, etc. – to focus on how 
best to manage their application’s runtime behaviour (i.e., its management logic) 
rather than focusing on the minutiae of running on a multi-cloud. In the classic 
MAPE-k model of autonomic systems [9], XCAMP implements and integrates 
both the Monitoring and Execution stages while placing the onus for Analysing and 
Planning on the user. The management logic (i.e., operational policies guiding the 
runtime behaviour of the managed application) is specified in the language of the 
user’s choice using their preferred environment and according to the methodology 
of their choice. We describe the architecture of this platform and the challenges in 
managing the complexity of the multi-cloud in Section 1.4.

The main contribution of this chapter is the creation, definition, implementation, 
and evaluation of a novel approach to application management on multi-clouds that 
confers autonomic properties on applications at runtime and that embraces devops-
style management and facilitates experimentation with diverse autonomic manage-
ment approaches (e.g., model based, rules/threshold driven, classic control, etc.) 
while abstracting away many of the low-level cloud programming details and nuis-
ances. An important use case for XCAMP will be as the management framework

---

1. [http://devopsdays.org](http://devopsdays.org)
3. In devops, developers collaborate with the operations team to build and manage services while in noops it is only the developers who do this.
4. In situations where there is no operations team devops is equivalent to noops. For the remainder of this chapter we will simply refer to devops.
5. The X is pronounced ‘cross’.
for the SAVI testbed to streamline the life-cycle management of applications on a novel cloud architecture and to simplify the process of deploying runtime management, facilitating research on this two-tier cloud system by non-cloud experts and students. XCAMP has already been presented in a hands-on tutorial at the SAVI Annual General Meeting (2013) in Toronto, Canada to a group of approximately 75 project members (i.e., students, researchers and industrial participants).

To demonstrate the effectiveness of our framework we have implemented a prototype. We use this implementation to demonstrate the feasibility of our approach with an experiment demonstrating the autonomic cloud bursting of a legacy application. Additionally, we have run XCAMP on a two-tier cloud architecture and we present a case study in which we diagnosed the root cause of a performance bottleneck observed on the SAVI testbed. Finally, an experiment measuring the throughput of our implementation, ensuring it is practical for managing large systems, is presented. The experiments (described in Section 1.5) effectively demonstrate the capabilities of our approach. Based on our implementation experience, we describe several ongoing challenges for management in the multi-cloud.

We close the chapter by positioning our work with regard to the state-of-the-art in Section 1.3 and offering concluding remarks (Section 1.8).

1.2 Background Concepts

Historically, a company owned a set of dedicated resources (e.g., a private data center) upon which their business applications were run. Typically there were many such applications and how these applications behaved in relation to each other was of paramount importance. Specifically, issues of ownership and access were critical (i.e., could application A run on machine Z between 5 and 8 PM EST). Further, an IT operations team was responsible for ensuring both the security and operations of the physical infrastructure and also with ensuring the effective functioning and security of all applications, including those developed in-house. Most applications ran on bare-metal (i.e., servers) and a ceiling existed on total available resources that was relatively constant (unless machine upgrades were performed or new resources were added to the data center’s footprint). Extensive work on management frameworks and methodologies supports these processes.

As described in the introduction, cloud computing is fast removing many of the standard management barriers that once defined the IT landscape. For example, the requirements to carefully plan for capacity is being eclipsed by the ability to programmatically launch virtual machines using a pay-as-you-go model (i.e., the IaaS cloud) as required. The responsibility of managing the physical infrastructure has been separated from the responsibility for managing applications. This affords significant flexibility, allowing for the fine-tuned management of resource acquisition and release, and dynamic configuration of managed applications. This

6Smart Applications on Virtual Infrastructure (SAVI) is a national research project in Canada: [http://savinenetwork.ca](http://savinenetwork.ca)

7The proposed architecture, implemented by the testbed, introduces a novel architecture (i.e., two tier cloud) where virtualized resources exist close to end users (i.e., the smart edge), allowing applications to access either low-latency resources near end users, or standard public cloud data centers (i.e., the core).

8That is, acquiring additional resources from a public cloud when a private cloud does not have sufficient resources to handle its workload [10].
new-found infrastructural flexibility has allowed developers (or, has allowed managers to push developers) to focus on business-level considerations and effective operational strategies (e.g., devops) as an alternative to focusing on highly optimized and tuned code design. The adoption of cloud computing by medium and large enterprises is expected to accelerate as a growing number of suppliers build additional datacenters, as virtualization technologies continue to improve, and as faster networking links provide high-speed connectivity.

While this development of technologies supporting the cloud continues to accelerate, the challenge of how best to manage applications deployed to the cloud remains unresolved. For example, in 2013 the Amazon.com website went down for longer than twenty minutes. One popular approach to the management of applications (including those on clouds) is referred to as autonomic computing. Autonomic computing was introduced as a way of dealing with the increasing complexity of systems. It is based on the concept of the autonomic nervous system, which in humans is responsible for the constant beating of the heart among other things. The outwardly observable behaviour of an autonomic application (i.e., one managed using this approach) is that of self-optimization, self-configuration, self-healing, self-protection (i.e., self-*) behaviour. This approach involves a key management component: the autonomic manager.

The autonomic manager is responsible for adjusting the behaviour of an application in response to both runtime and management policy constraints. More precisely, an autonomic manager’s function can be decomposed into a loop composed of four main phases: monitoring, analysing, planning and execution (i.e., the MAPE-k loop). In the monitoring phase, the autonomic manager monitors the performance of the application (and possibly the environment, etc.). In the analysis phase, the autonomic manager analyses this data to build up an understanding about what possible strategies to apply to improve the application’s state. In the planning phase, the autonomic manager selects a strategy from among the possible strategies. In the execution phase, the chosen strategy is implemented.

A key characteristic of autonomic computing is automation. This is also true for devops. Devops and related approaches are used by major industry trend-setters (e.g., Amazon and Netflix). Complementary to devops is the process referred to as continuous deployment in which the release cycle is shortened from months to days (or even less). For example, Amazon.com deploys a release every 11.6 seconds. Although traditional applications are faster to develop and deploy and easier to manage in clouds, autonomic applications are still difficult to design, implement, and deploy, and still require substantial knowledge and resources. A goal of this work is to make development and deployment of autonomic applications easier. XCAMP mechanisms for automating the life-cycle (i.e., deploy, manage, undeploy) of not only the application, but also the management logic responsible for autonomically managing it.

The first step in cloud adoption is often transitioning an on-site datacenter into a private cloud. However, private clouds, while providing many of the benefits of a general cloud (i.e., on-demand resources) lack many of the economies of scale inherent in public clouds such as massive scale and freedom from equipment storage/
maintenance/personnel costs, etc. As a result, both hybrid public-private clouds and cross-provider deployments are becoming more common. It is well known that one of the biggest challenges of constructing both hybrid clouds and/or the multi-cloud is the bridging together of multiple infrastructures. Difficulties include but are not limited to abstracting away the details of the various provider-specific syntaxes [13], unifying/normalizing the various pricing models [14], providing seamless monitoring across potentially quite disparate provider domains [15], ensuring data ownership, privacy, locality, security, etc. This motivates the need for abstraction of the low-level operations on the multi-cloud, a need XCAMP is designed to meet.

1.3 Related Work

The notion of on-demand systems existed well before the advent of cloud computing [16, 17]. Noticing the scale and increasing complexity of systems, IBM introduced the notion of autonomic computing [9] that popularized the notion of a MAPE-k loop and self-* functionality. The concept of autonomies has also been considered by [18, 19]. These concepts can be understood as precursors and/or progenitors in one way or another of the current notion of the cloud.

Managing resources in this emerging cloud environment is a significant challenge; Jennings and Stadler enumerate key aspects of this challenge, including: “the scale of modern data centers; the heterogeneity of resource types and their interdependencies; the variability and unpredictability of the load; as well as the range of objectives of the different actors in a cloud ecosystem” [20]. As the cloud has begun to take shape, several tool-kits and frameworks have been introduced as possible approaches to addressing the challenge of managing resources while extending the cloud’s capabilities. Some well known examples of these include Reservoir [21], OPTIMIS [22], Aneka [23], and VDC Planner [24]. Often, these approaches include notions of federation, multi-cloud, hybrid cloud, etc. However, they are all devised from a more traditional perspective in which a deployment must be carefully designed and optimized in advance so that it may negotiate a correct SLA to ensure its requirements are met. Where these frameworks are forward-looking and require complex architectural components we chose instead to focus on the cloud as it is presently available. This design choice allows us to help bring new users to the cloud and facilitates experimentation with various approaches to the design and implementation of management logic (e.g., model-based, rules and/or threshold driven, classic control, etc.). Our focus was on facilitating management of applications by the developers not on how to manage the cloud from the perspective of an infrastructure provider.

An important aspect of facilitating a multi-cloud involves the notion of a broker [8, 25]. A broker acts to facilitate resource acquisition and release on behalf of a client application in response to their dynamic requirements at runtime. While in some cases, as demonstrated in this chapter, the management logic suffices to determine from where to obtain and/or release resources to; in other scenarios in which multiple potential competing providers exist a broker provides a logical component to obtain/release the best selection of resources as required. Therefore we envision future integration between managers and brokers. The broker will be responsible for resource acquisition/release while the manager will be responsible for application management tasks.
1.4  X-Cloud Application Management Platform (XCAMP)

The design of XCAMP is based on the MAPE-k \([9]\) loop, with framework components and developer-specified components working in collaboration to perform MAPE-k-based management of an application. The monitoring and executing portions of the loop are performed by framework components, while the analysis and planning portions are done by developer-specified management logic. This separation of concerns guides runtime operations and is presented graphically in Figure 1.1a. XCAMP was designed to work at multi-cloud scale (i.e., massive application deployments of thousands of nodes) and is able to support multiple application deployments simultaneously. XCAMP leverages a stream processor paradigm to achieve scalability, fault-tolerance, and reliability, and to provide a useful abstraction of streams (long sequences of records) to transfer metrics, key performance indicators (KPIs), and in general knowledge among the components, with each new tuple processed in transit by the various components. The following sections will provide an overview of the XCAMP architecture in terms of the MAPE-k loop and then delve more deeply into its components and the abstraction features of the platform. First, two usage scenarios will illustrate the use of the XCAMP platform, one focusing on the impact on a single application, and the other from the perspective of a service provider.

1.4.1 Usage Scenarios

In this section we introduce two illustrative usage scenarios for XCAMP.

**Scenario 1, Hybrid Clouds**: Company A would like to deploy an application to their private cloud. However, they are constrained by a lack of resources to support it during peak periods of demand. They wish to create a hybrid cloud, using public cloud resources when private resources are exhausted, with resources being added and removed autonomically based on demand.

- **Preparation**: After deploying XCAMP to their private cloud\(^{12}\), they would register both their private cloud and Amazon Elastic Compute Cloud (EC2)\(^{13}\) with the platform. Next, they would create a deployment document that describes the layout of their application on cloud resources (i.e., this includes describing nodes, images, services and communication links between services). They would capture their desired autonomic behavior in rules (e.g., one rule might be *when resource utilization in the web tier exceeds 60%, add a node to the private cloud, unless resources are exhausted, then add a node to the public cloud*). The rules use the terminology defined in their deployment document (e.g. *web-tier*), and can reference any metric captured by XCAMP. This set of rules is called *Management Logic* throughout this chapter, and represents the management policies to be enforced, as implemented by an application capable of accepting monitored metric values at a specified URL, making management decisions based on this stream of metrics, and returning actions to effect change in the deployed application as needed. This web-based

\(^{12}\) An automated installation using from 1 to 5 VMs

\(^{13}\) [http://aws.amazon.com/ec2/](http://aws.amazon.com/ec2/)
application is implemented in whatever language the developer prefers, and is deployed automatically by XCAMP into an appropriate container (e.g., Apache Tomcat).

- **Deployment**: The administrator then submits their deployment documents together with application and Management Logic to the system along with any additional automation scripts (i.e., to setup a database). XCAMP auto-
matically instantiate cloud resources and dynamically builds the application according to the given descriptions. Upon instantiation, the platform automatically begins capturing metrics from all configured resources, and feeding this stream of metrics to the Management Logic’s defined URL.

- **Runtime Management**: As the Management Logic receives metrics from XCAMP, it returns (as-needed) actions that are realized by XCAMP. In this scenario, the company would author their Management Logic application to detect increases in demand (as reflected by increases in utilization) and in response add application servers first on the private cloud, then on Amazon EC2 when private resources are exhausted. XCAMP handles the process of adding resources, including creating instances (of the specific image) from the correct cloud provider (private cloud, EC2), dynamically installing the correct packages, instantiating the correct services, and connecting these new nodes within the application environment topology (i.e., adding them to the front end load balancer and pointing them to the database). Similarly, as demand recedes, these resources can be automatically released and decommissioned.

**Scenario 2, Edge-Core Clouds**: The SAVI two-tier cloud is made of edge nodes, close to the end user, and core nodes, located in a big data centre. The architecture is meant to support low latency and high bandwidth applications. SAVI administrators want to provide a management service to the users of a testbed implementing this SAVI cloud architecture. The XCAMP platform must enable users to deploy and manage their applications while accommodating a broad range of practical experience (from novice to expert) with regard to deploying and/or managing applications on the SAVI cloud.

- **Preparation**: The administrators must deploy XCAMP to their two-tier cloud architecture, then provide the XCAMP front-end URL to their users. Administrators can decide where to place the initial deployment, on the edge or core nodes, and then author a deployment descriptor.

- **Deployment**: SAVI researchers submit their jobs (i.e., the application and Management Logic) through the web interface or RESTful API. XCAMP deploys the application on the edge and core nodes provisioning at the same time the connectivity among components.

- **Runtime Management**: XCAMP ensures all monitored details about a user’s deployed application environment is collected and routed to the user’s Management Logic, and accepts all commands issued in response, translating them to low-level actions and executing these actions. The management logic can act on application components on edge or core nodes. Typical actions includes scaling out/in on edge and core nodes, live migration of VMs for load balancing, etc. A key design pattern of XCAMP is its utilization of stream-processing that facilitates its ability to scale to massive size and support large number of nodes while collecting massive numbers of measurements about runtime performance and external monitored details. The entire SAVI testbed shares this single platform, avoiding duplication which results in high utilization efficiency.
1.4.2 MAPE-k Loop View

The key XCAMP components and their position in the MAPE-k loop are presented in Figure 1.1c. In this section we will describe these various components and their contributions to the management of a deployed application on the multi-cloud.

The **Information Aggregation Service** is the main interface for gathering information about the deployed applications, environment and from external sources (e.g., Twitter, CloudyMetrics [14], CloudHarmony[^1] and others). Collected information is streamed to the **Notification Engine** which is used to forward an augmented stream of metrics to the various **Management Logic** components. In one path, data traverses the **Abstraction Engine**. Given information from the knowledge store that describes all existing deployed applications, each metric is translated to a more abstract form based on the terminology defined in the deployment document (e.g., an IP address is translated to a unique identifier that is marked as belonging to web-tier).

In a second path, data traverses the **Plugin Engine** (optionally first passing through the abstraction engine) where additional processing is applied to the data stream. For example, aggregation may be applied to individual server metrics constructing tier-specific information (e.g., mean CPU utilization per cluster) or the archive of metrics hosted by the Information Aggregation Service may be queried to produce metric trends. The platform provides several plugins (for example, calculating the cost of a deployment based on information from CloudyMetrics); the user may add their own. Information that leaves the plugin engine is specific to a given application, either using information from the abstraction engine or from the user-supplied plugins.

The **Management Logic** represents developer-specified management directives (e.g., management policies [26]) designed to guide the runtime behaviour of the application. The XCAMP framework does not place any restriction on how the Management Logic is expressed/implemented; it is run within a sandboxed container, on its own virtual machine. The Management Logic represents a combination of both the Analyze and Plan components of the MAPE-K loop. It offers a URL to which metrics are submitted, and responds with JSON-formatted actions that are passed to the Execution phase. Each developer uses best-practices for filtering requests for their chosen platform (e.g. Java EE Filters) to decide which metrics reach the web application logic. A Management Logic component is responsible for managing its own data store if required. A partial excerpt from a Java-based implementation of Management Logic is presented in Figure 1.2.

The **Execution Engine** and the **Deployment Service** implement the Execution component of the MAPE-k loop. The Execution Engine accepts requests for changes from the Management Logic and converts this into high-level workflow statements. These statements are forwarded to the Deployment Service, which executes a set of lower-level workflows to implement the requested changes to the application’s deployment and/or configuration. For example, the Management Logic might request an additional resource be added to its web-tier; the Execution Engine translates this request to a parameterized call to the deployment service which creates and provisions the node and re-configures the load-balancer. The components collaboratively maintain a knowledge base of system state through the **Knowledge**

Figure 1.2: Sample code (with exception handling omitted to simplify readability) for the cloud bursting Management Logic implementation is presented. This Management Logic is implemented as a Java servlet. The `doGet` method is called from the monitoring components of the XCAMP platform with updates about all relevant monitored metrics and the response is either empty or carries an action to be performed by the XCAMP Execution Engine. On line five an update about load_one is processed for a particular node in which the metric `METRIC_NAME` (i.e., load_one) for the node `SOURCE` of the application topology is updated with the value `VALUE`. The management rule presented on line eight is one of the four rules used to implement the elastic bursting strategy and can be stated informally as follows: \textit{IF the mean load for the application server tier is less than a threshold and the size of the application server tier on the public cloud is greater than MIN_PUBLIC_SIZE THEN scale down the public footprint of the application server tier by SCALING_INCREMENT.} The `ActionGenerator` class on line 14 is used to generate and send JSON messages to indicate what action the execution action should take (e.g., line 18).

Store component which stores data about the historical, current, and predicted future state of the system.

1.4.3 Deployment View

The process of application deployment requires that the developer submit a declarative deployment document \cite{9} that describes the application pattern to deploy \cite{10}, a deployable version of their application (e.g., a WAR file), and a Management Logic web application (e.g., a WAR file). The submission component (not shown) passes this information to the deployment service, which deploys the application in accordance with the deployment document (using user-supplied credentials) and registers the deployment in the Knowledge Store. The Management Logic applica-
tion is deployed, and is automatically registered with XCAMP to receive pertinent information for its associated application. The deployer may specify external data sources from which information should also be retrieved. Application-level metrics are submitted to the Information Aggregation Service and will be available to the Management Logic.

The result is the automatic collection and pushing of well-formatted, high-level, abstract, consistent metrics to the Management Logic. Using whatever approach and methodology the developer prefers – ad hoc Java code, a web interface wrapper to an existing management system, etc. – the analysis and planning steps are completed. If actions are required, a JSON message is passed to the Execution Engine, where it is de-abstracted and passed to the deployment service to modify the running deployment.

Once an application is deployed using XCAMP, its structure will be similar to those of the applications presented in Figure 1.1. Functionally, an application deployment represents a complex graph of an application in which nodes are virtual machine instances (VMIs) running on the various cloud providers and edges represent communication channels between these nodes. XCAMP deploys a management agent to each VMI in an application. This agent is responsible for transmitting collected monitored data to the Information Aggregation Service and for modifying configuration settings of the installed application stack, the operating system, and/or altering the set of installed applications on the VMI in response to commands from the deployment service.

To facilitate operations, XCAMP communicates directly with the various cloud provider APIs (e.g., AWS, Openstack, etc.), which allows it to perform operations like adding and removing instances, and to collect metrics from the provider when available. XCAMP also monitors data from sources other than cloud resources and further passes it to the Management Logic; for example, data from Twitter, CloudyMetrics, or CloudHarmony can be passed to Management Logic to assist in decision making. For example, should there be a failure of a region in AWS, XCAMP will receive this status update. This data can be utilized by the Management Logic in order to make decisions about where to deploy nodes of the application. This might include transposing application servers to alternative cloud providers on the fly or simply to avoid launching new VMIs in affected regions.

### 1.4.4 Information Abstraction in XCAMP

A key contribution of XCAMP is the abstraction of low-level details that differ among various cloud providers, and a common metrics format. To illustrate how we hide complexity in XCAMP from the management logic, consider the following example of adding an instance to a deployed application.

After the Management Logic determines that an instance must be added to the application server tier of a deployed application, it sends a JSON-formatted message to the Execution Engine saying *There should be five servers similar to web server A in cluster my-web-tier*. The Execution Engine translates this declarative request for resources into a high-level workflow, which is passed to the Deployment Service with associated information (e.g., which application deployment to modify). The Deployment Service translates this into a low-level workflow as follows. First, the Deployment Service determines upon which cloud provider Cluster my-web-tier is running. This allows it to connect to the correct cloud provider. It then requests
two instances of the same configuration as web server A (determined by the deployment document, e.g., m1.large) and deploys the management agents on both instances. The management agents are then instructed to deploy an identical software stack with the same configuration as web server A. After the instances are fully configured and ready the agent will begin streaming monitored data to the Information Aggregation Service, which will ultimately inform the Management Logic of the successful addition of resources to the deployed application.

1.4.5 Management Logic

Much work has been done in the domain of distributed systems management (see for example the proceedings of IEEE/IFIP NOMS, IFIP/IEEE IM, CNSM). Policy-based management \cite{26, 29, 30} represents an approach to management in which management actions are decoupled from management logic and in which the management logic is interpreted at runtime. This affords a flexible management paradigm in which as management logic changes, policies may be altered thus facilitating the design of elegant autonomic systems. Often, a policy specification language \cite{31, 32, 33} is used to encapsulate the rules that govern the behaviour of the system. While we embrace the need for specification languages, especially in the context of large distributed systems and network management, we feel that their utility is tightly coupled to the actor who is tasked with using them and the specific environment in which they are used. In devops, we feel freedom should be given to the developers to do it their way and to take advantage of all the intimate details that they possess with regards to the inner workings of the application that they are managing. Unlike in traditional management situations where system administrators, operations teams and/or business people require a mechanism for automating the control of their systems that is semantically clear to them and does not place much emphasis on programming capabilities. Developers have an entire arsenal of tools, libraries and methodologies for ensuring that a system is functionally correct. These can be embraced to ensure the non-functional requirements of a system are being met as well. Due to their experience with programming languages (and likely lack of experience with DSLs like policy specification languages), use of programming languages may be a preferred approach for specifying management logic. Further, our proposed approach does not in any way preclude the use of an existing policy specification language/PBM solution, which could be readily employed as the Management Logic component.

1.5 Implementation

To demonstrate the feasibility of our approach, we authored a proof-of-concept implementation of XCAMP. We leveraged existing libraries and frameworks where possible to allow us to focus on the abstraction task. Monitoring components are built on the Misure \cite{15} extensible, distributed, scalable monitoring system. Due to its central importance in XCAMP we provide a brief overview of it in Section \ref{monitoring}. The Information Aggregation Service, Abstraction Engine, Plugin Engine, and Notification Engine were written as elements that used the stream-processing paradigm central to Misure to communicate and scale horizontally.
Execution is provided by the Execution Engine, which like the other engines is built on Misure; and by the deployment service, for which we used a customized version of the Pattern-based Deployment Service (PDS) developed by our team. Similar to Misure the PDS plays an important role in XCAMP and so we elaborate on it in Section 1.5.2. The Execution Engine connects to the deployment service via a RESTful API.

Analysis and Planning is provided by Management Logic applications, one per deployed application, running in a Java EE container (Tomcat) on a dedicated VMI. The responsibility for authoring the Management Logic application rests with the application developer/deployer; they submit WAR files for deployment. Any container is adequate for this purpose; others could be added with a straightforward extension to the current implementation.

1.5.1 Misure

In previous work, we defined a set of requirements for monitoring in heterogeneous federated clouds, defined a suitable architecture built on stream-processing, and implemented a prototype solution. Based on an enhanced publish-subscribe pattern, the design and implementation allow for the gathering of metrics at any level (system, application, etc.) from disparate sources like Ganglia, SNMP sources, Amazon Cloudwatch, and various web APIs. These streams of metrics are transformed (aggregated, annotated, split, etc.) in transit, and published to interested users as live streams (push-type). Metrics are also persisted to long-term storage which can be queried via an API (poll-type). The prototype was evaluated on public clouds and found effective at handling metrics at scale with low infrastructure cost.

The core abstraction underlying Misure is stream-processing, in the family of complex event processing; as an abstract concept, it refers to the generation, manipulation, aggregation, splitting, and transformation of data organized in a long sequence of records. One example is Storm, a Twitter open-source project, on which Misure is built. Storm is billed as a “distributed, scalable, reliable, and fault-tolerant stream processing system”, and can be used for stream processing, continuous computation, and distributed RPC.

One of the key features of Storm is the effort to manage the complexity of distributed computation on realtime data entirely behind the scenes. This includes guaranteed message processing, aggressive resource management (garbage collecting defunct processes), fault detection and task reassignment after failure, efficient and scalable message transmission, streams that consist of any data (serialization occurs behind the scenes), and local development environments for debugging. Storm also allows components to be implemented using many programming languages. Storm is parallel and distributed; there is no central router and no intermediate queue. It is designed to scale horizontally, and has been deployed at scale processing large Twitter data sets.
Figure 1.3: A sample deployment document written in an XML-based DSL. This particular document describes a deployment on six nodes in total (i.e., one web-balancer, four web-hosts, and one database-server).

1.5.2 Pattern-based Deployment Service

The pattern-based deployment service, PDS [27], emerged out of the need to simplify the process of deploying complex multi-tier applications to cloud environments and further to adapt them at runtime (i.e., dynamically add/remove nodes to an existing application topology). Specifically, the notion of describing declaratively what you want versus how to achieve it appealed to us. Further, the use of patterns is quite powerful in they allow for simplification, re-use and sharing. The PDS hides the low-level details about how to deploy services to any cloud provider, making it quite useful in the context of multi-cloud. Specifically, the PDS facilitates application topologies to be described, deployed and adapted across multiple cloud providers. The PDS has been used successfully on EC2 and Openstack and has support for all Fog compliant cloud providers. Further, the PDS has been open-sourced and is available for download and contribution [19].

With PDS, a user describes the “pattern” of their application in an XML-based Domain Specific Language (DSL). This DSL is quite easy to understand and is

---

18 http://fog.io/about/provider_documentation.html
19 https://github.com/ceraslabs/pattern-deployer
composed of several key elements which can be grouped as functional elements (e.g., Topology, Node and Service) and syntactic sugar (e.g., instance templates, Container, and num_of_copies). A sample deployment document written in this XML-based DSL is presented in Figure 1.3.

1.6 Experiments and a Case Study

Using our implementation, we performed two experiments (Sections 1.6.1–1.6.3). In the first, we demonstrate the process of implementing Management Logic and using it to manage an application facing changing workloads; in the second, we show early performance results demonstrating that a Management Logic component can handle millions of metrics. We also used this implementation in a case study demonstrating how researchers running on an experimental testbed can more easily perform complex experiments repeatedly to obtain meaningful results (§1.6.4).

1.6.1 Experimental Setup

Experiment 1: For the managed application, we used a sample Java EE web application that accepts requests; connects to a database (mysql) to perform selects, inserts, or updates; and returns a response. We defined a declarative deployment document (see Section 1.4.1) with a database server, a load balancer, and a cluster of web application servers initialized to a single instance running in the private cloud and no instances in the public cloud.

We authored the application’s Management Logic, see Figure 1.2 for an excerpt of this code, as a second Java EE web application, implementing the RESTful interface defined by the platform to receive new metrics and information about resources deployed. The Management Logic collected the 1-minute load averages for each web application server, calculated an average, and requested additional resources when a configurable threshold was surpassed. Resources were released when the average fell beneath a second configurable threshold. Given the limited capacity of private clouds, after two instances are running on the private cloud, the manager requests resources from the public cloud. To limit churn, a refractory period was introduced (as a configurable parameter that could be changed on the fly through the RESTful interface): 10 minutes between adding nodes, and 5 minutes between removing nodes. Aside from features to allow run-time configuration of various parameters, the Management Logic consisted of 65 source lines of code.

The PDS was deployed to an Amazon EC2 m1.small instance. We created a deployment package including the web application WAR, its Management Logic WAR, and the various keys and credentials required to provision instances on the clouds in our topology, and submitted this package to the PDS. The Management Logic ran on a t1.micro instance; web application servers were deployed to a local Openstack installation running on a dedicated IBM Bladecenter in a university data center, with a 100Mbps uplink. An openstack.small instance was defined with 2GB

\[\text{Load average refers to the number of processes ready for CPU time on average over some period of time, 1 minute in this case.}\]

\[\text{Clearly there is substantial room to improve this algorithm; the focus is on the enabling platform not the adaptive scaling algorithm employed.}\]
of RAM and 1 virtual CPU. The public cloud deployment – if necessary – was to m1.small instances on Amazon EC2.

The XCAMP implementation ran on a four-core cluster in Amazon EC2. Ganglia monitors acquired the metrics from each machine and passed them to the Information Aggregation Service.

Finally, an Apache JMeter test plan was used to generate load. The workload was two-thirds read requests (e.g. browse catalog), one-sixth write requests (e.g. checkout), and one-sixth update requests (e.g. modify user profile). The design was to peak workload at 120 simultaneous threads sending requests as quickly as possible, launching in four groups of thirty threads, each ramping up over a 5-minute period. The first group launched at start time $t$ minutes; the second at $t + 10$, the third at $t + 25$, and the fourth at $t + 33$. Peak workload was maintained until $t + 90$, when the fourth group was terminated, followed by the second group at $t + 100$, and the final two groups at $t + 120$. This plan was executed by an m3.xlarge (quad-core) instance in Amazon EC2.

**Experiment 2:** Using the same implementation and components, we examined the performance of the Management Logic when run on three different instance sizes in Amazon EC2 (t1.micro, m1.small, and m1.medium) to assess the scalability of this approach. We created simulated metrics and submitted them via the RESTful API as quickly as the service could handle them for a one-hour period. The mix of metrics was proportional to reality, with many requests being irrelevant to the actual decision-making. Our primary question was whether it would be necessary to autonomically scale the Management Logic Container as a cluster for large topologies.

### 1.6.2 Results

**Experiment 1:** Fig. 1.4 illustrates what happened during the experiment, showing the addition and removal of instances, the size of the workload, the average load over all deployed resources, the average response time, and the total throughput. The deployment began with a single private instance. This was sufficient for the first workload group but shortly after adding the second workload group the autonomic manager detected load average greater than 1.0 (Fig. 1.4a). A private cloud instance was requested (light red band). There are brief spikes in response time (up to 4 seconds) when the node is added to the balancer manager and when the node is first enabled and receives its first requests which are not shown due to the smoothing (for readability). The two private instances are sufficient to handle 60 workload threads, but not 90 where a third instance is required. This instance is requested from Amazon EC2 (light orange band).

As the experiment continues, workload continues to increase and the load average remains high. Amazon m1.small instances are substantially smaller than openstack.small instances; a total of five are required (added as soon as possible given the refractory period) to meet the generated workload. After the activation of the final public instance, load average settles at around 1 and remains there,

---

22 [http://ganglia.sourceforge.net](http://ganglia.sourceforge.net)
23 [http://jmeter.apache.org](http://jmeter.apache.org)
24 Note that instances running on this Openstack installation using KVM use a virtual CPU and report load averages differently, counting processes waiting in a queue and NOT running processes.
Figure 1.4: Measurements from the scaling experiment, first adding private instances then bursting to the public cloud. The stacked graphs show the instance counts, with the blue line representing throughput (smoothed to be more readable).

providing stable response time and maximum throughput. Once the workload decreases, the additional instances are gradually removed – first the public instances, then the private instance (at the end of the experiment).

**Experiment 2:** Fig. 1.5 presents the results of the scaling, showing the total throughput and average response time achieved by the three instances running the straightforward bursting adaptation policy. A gradual ramp-up was included in the load generation. The t1.micro instance is specified for only periodic or bursting workloads, not for sustained load; this is evident in the results as performance varies dramatically. There are several drops in throughput, due to either other running tasks competing for resources or the variation inherent in the public cloud [36] which is most noticeable with smaller single instance sizes. The t1.micro response time results make it difficult to see corresponding degradation in response time.

1.6.3 Discussion

**Experiment 1:** We demonstrated the ability to use a standard Java EE web application with a simple adaptation policy written in a programming language familiar to the original application developer using the common RESTful API pattern. There is room to improve the Management Logic; for example, to better handle the time required for a new instance to become active and handle requests.
Figure 1.5: Measurements from a scaling test measuring the throughput of our implemented Management Logic on three Amazon EC2 instance sizes.

While designing the Management Logic, we considered several metrics as the basis for adaptation. We have been aware of the limitations of existing monitoring tools on public clouds for some time, but our trials with CPU utilization metrics and load averages demonstrated the unreliability of these numbers. The individual metrics for each of the seven instances launched in Experiment 1 varied per-cloud. The load averages from OpenStack were zero even when the machine was clearly loaded; it was only when overloaded that they would produce higher load averages, which resulted in slower reactions from the Management Logic. The data from EC2 had higher peaks, particularly during bootstrapping. More notably, despite high load averages, they rarely exceeded 20% CPU utilization (Fig. 1.6b).

In contrast with 1-minute load averages (Fig. 1.6a), the 15-minute load average offers a better understanding of the overall trend of the system. Fig. 1.6c shows this load average for each instance. All of the managed instances trended toward a load average of 1.0, the target set by our Management Logic. Much of the difficulty in achieving this load average on EC2 instances hinged on load incurred during bootstrapping. This suggests that launching from machine images with the required software pre-installed is important to effective adaptive management.

**Experiment 2:** The performance numbers measured indicate an ml.small instance running our Management Logic could process over 270,000 metrics per minute; collecting the standard 18 Ganglia core metrics once per minute suggests an ability to manage 15,000 active instances. This suggests there is currently no need to autonomically scale a cluster of containers. There is no strict bound on complexity for alternative Management Logic applications, and so this need may arise in the future if computation-intensive planning and analysis is performed. The scalability of Misure has been discussed previously [15]; it is similarly capable of handling thousands and even millions of metrics.

### 1.6.4 Case Study

One of the goals of XCAMP is to make systems management painless for developers. This case study illustrates this ability in action. It was noted that in practice, when several SAVI users deployed applications simultaneously using XCAMP, they
Figure 1.6: Performance measurements from instances involved in the autonomic scaling experiment.

We observed a major degradation in performance of the SAVI testbed\textsuperscript{25} We used XCAMP as the platform for a series of experiments to explore this phenomenon to contribute to the ongoing improvement of the testbed.

**Initial exploratory runs:** Using XCAMP’s deployment service, we deployed a three node Java EE application to the SAVI testbed (note that all three nodes are deployed simultaneously by default). Once deployed, we dynamically added an additional node to the application topology. Once the scale out operation had completed we removed this additional node by scaling down. Finally, we undeployed the application. We kept measurements of how long each stage of this process took.

\textsuperscript{25}The testbed implements a two-tier cloud extending OpenStack, with a single core and 7 edges distributed across Canada.
We conducted variations of this experiment with various numbers of simultaneous application deployments (1, 2, 5, and 10), each deploying three-node Java EE applications (for between 3-30 VM instantiations). Each experiment configuration was run three times. The mean timing results (with standard deviations) are presented in Figure 1.7a. We observed that the Deploy stage is the slowest of the four and that it was most impacted by the number of concurrent users; scale-out, which is like deploy but with one instance instead of three, was also impacted.

**Examining the deployment stage**: We decided to explore the Deploy stage more carefully by examining the two phases: downloading required files from the PDS on the internal network, versus downloading and installing software packages from an external package repository. We also performed the same experiments on Amazon EC2 in order to use the results as a comparator. Figure 1.7c shows a linear increase in download time as the number of concurrent users increases for both EC2 and the SAVI testbed. However, software installation on EC2 appears to be constant no matter how many concurrent users there are while on the SAVI testbed a dramatic increase is observed as the number of concurrent users increases. We hypothesized it might be the network and/or a disk IO-related problem. The network-bottleneck hypothesis is that additional concurrent users are creating traffic on the network, causing congestion or bandwidth-cap related issues. The IO-bottleneck hypothesis is the increased number of VMs running on a single physical host and performing random read-writes overextends disk resources. We know from observation that CPU and memory utilization are not excessive, so we did not create additional resource-contention hypotheses.

**Evaluating hypotheses**: To confirm or reject each hypothesis, we designed a final experiment. We created a full image containing all required packages for the application. We compared the time required to deploy this image, versus the time required to deploy a standard Ubuntu image and bootstrap it (i.e. download and install all required software packages from a central repository). A complete image has similar bandwidth requirements, but can be written to disk with sequential writes (versus random read-write) thus reducing IO load. If deploying the full image is faster than bootstrapping, we would regard the IO-bottleneck hypothesis as confirmed. The results are presented in Figure 1.7b. Notice that the full image outperforms the standard image, suggesting the presence of an IO-bottleneck.

Using XCAMP in this case study allowed us to easily run and monitor a variety of experiment configurations, systematically and repeatedly, to collect and present evidence of system performance issues in the SAVI testbed.

### 1.7 Challenges in Management on Heterogeneous Clouds

Based on our experience designing, implementing, and testing a multi-cloud adaptive system, we offer the following reflections on the particular challenges that apply to adaptively managing heterogeneous clouds.

**Heterogeneous monitoring systems**: Many cloud providers offer monitoring services to provide information about the performance of provisioned resources; these systems vary significantly, and typically require relatively detailed information to query (e.g. instance IDs for Amazon EC2). The state of monitoring in private clouds is even more varied, with various solutions deployed based on each orga-
Figure 1.7: Various experiment results for the case study exploring performance of the SAVI two-tier cloud testbed.

Rapid reaction: Automated management requires current and accurate monitoring data. The ability of an aggregating monitoring service to meet this requirement depends on the timeliness of metrics received from third-party monitoring systems being aggregated. The automated manager can be run on the public cloud, which introduces more delays outside of the control of the monitoring system. It remains an open question how to best ensure timely decisions are made.

Inaccuracy in traditional monitoring techniques: It is generally understood that virtualized resources offer more variable performance than bare-metal resources, and the variance in the performance of Amazon EC2 instances has been benchmarked [36]. However, it is less understood that standard monitoring techniques will report inaccurate information that can mislead adaptive managers.

In a public cloud environment, the desire to sell fractions of a CPU’s processing power have removed the meaning of many of these standard metrics. For example, Amazon EC2 configures instances using a measure called Elastic Compute Units (ECU), which they document as a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon proces-
22 PERFORMANCE MANAGEMENT AND MONITORING

sor. 1 ECU is approximately 40% of a single core of a Intel Xeon CPU E5430 @ 2.66GHz, a common processor in the first-generation Amazon infrastructure. The default instance size, small, is 1 ECU; the hypervisor enforces this 1 ECU limit. However, Xen’s paravirtual mode offers limited ability to abstract the processor for performance reasons, so instances and Linux kernels running on instances perceive a full core available to them. Xen enforces the limits by refusing access to the CPU if the allotted quota has been used, which the Linux kernel reports as steal. The exact time spent in steal may vary over time, and in any case can only be measured when the system is operating at capacity. An idle machine will report 100% idle time, giving no indication of the actual limits on the CPU. It is not trivial to calculate the actual load on a machine reporting 20% user and 80% idle.

1.8 Conclusion

This chapter introduced a framework for managing the life-cycle of applications in multi-cloud environments and for conferring autonomic properties on them at runtime. The framework allows the specification of the management logic, its deployment and instantiation, and its execution alongside the managed application. The Management Logic runs in a container that is seamlessly connected to XCAMP’s monitoring and execution engines. XCAMP provides application developers with effortless access to monitoring sensors, third-party data sources and actuators (i.e., the monitoring and executing stages of the MAPE-k loop) from across the multi-cloud, while placing control of both the analysis and planning stages in their hands and allowing them to express their management policies in their own vernacular and harnessing all their personal expertise.

We validated XCAMP in multiple ways. First we demonstrated the ability to elastically cloud burst a legacy Java EE application from a private cloud to a public cloud and reported our findings and our experience in automating applications in multi-cloud environments. Additionally, we demonstrated a capability of the XCAMP framework to facilitate the diagnosis of a bottleneck on the SAVI (i.e., two-tier cloud) testbed. We demonstrated, through experimentation, the capabilities of our design to scale to large size and for our autonomic manager (i.e., Management Logic) to process massive numbers of metric updates per minute. Finally, we presented a hands-on tutorial to a group of approximately 75 SAVI members at the 2013 Annual General Meeting in Toronto, Canada and received positive feedback from the participants.

The XCAMP platform will provide a useful middleware upon which to base much future research for both the SAVI project and other areas of multi-cloud research. By allowing developers to harness their vast skill sets, different approaches to management can be considered with ease and this is a critical benefit we provided through the introduction of XCAMP.

Acknowledgment

This research was supported by IBM Centres for Advanced Studies (CAS), Natural Sciences and Engineering Council of Canada (NSERC) under the Smart Applications on Virtual Infrastructure (SAVI) Research Network, and Ontario Research Fund under the Connected Vehicles and Smart Transportation (CVST) project.
REFERENCES


REFERENCES


[34] D. Luckham, The power of events: an introduction to complex event processing in distributed enterprise systems. Addison-Wesley, 2002.