A Framework for Developing DevOps Operation Automation in Clouds using Components-off-the-Shelf

Yar Rouf
York University
Toronto, Ontario, Canada
yarrouf@my.yorku.ca

Joydeep Mukherjee
York University
Toronto, Ontario, Canada
jmukherj@yorku.ca

Marin Litoiu
York University
Toronto, Ontario, Canada
mlitoiu@yorku.ca

Joe Wigglesworth
IBM Canada
Markham, Ontario, Canada
wiggles@ca.ibm.com

Radu Mateescu
IBM Canada
Markham, Ontario, Canada
mateescu@ca.ibm.com

ABSTRACT

DevOps is an emerging paradigm that integrates the development and operations teams to enable fast and efficient continuous delivery of software. Applications and services deployed on cloud platforms can benefit from implementing the DevOps practice. This involves using different tools for enabling end-to-end automation to ensure continuous deployment and maintain good Quality-of-Service. Self-Adaptive systems can support the DevOps process by automating service deployment and maintenance without manual intervention by employing a MAPE-K (Monitoring, Analysis, Planning, Execution- Knowledge) framework. While industrial MAPE-K tools are robust and built for production environments, they lack the flexibility to adapt large applications on multi-cloud environments. Academic models are more flexible and can be used to perform sophisticated self-adaption, but can lack the robustness to be used in production environments. In this paper, we present a MAPE-K framework that is built with existing Components-off-the-Shelf (COTS) that interacts with each other to perform self-adaptive actions on multi-cloud environments. By integrating existing COTS, we are able to deploy a MAPE-K framework efficiently to support DevOps for applications running on a multi-cloud environment. We validate our framework with a prototype implementation and demonstrate its practical feasibility by a detailed case study done on a real industrial platform.

ACM Reference Format:
Yar Rouf, Joydeep Mukherjee, Marin Litoiu, Joe Wigglesworth, and Radu Mateescu. 2021. A Framework for Developing DevOps Operation Automation in Clouds using Components-off-the-Shelf. In *Proceedings of the 2021 ACM/SPEC International Conference on Performance Engineering (ICPE '21), April 19–23, 2021, Virtual Event, France.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3427921.3450235

1 INTRODUCTION

DevOps[14] is a popular practice that enables the development and operation teams to continuously coordinate and collaborate with each other. By adopting DevOps practices and tools, it is possible to build, deploy and manage services and applications seamlessly while providing good Quality-of-Service (QoS) to the end users. With more and more commercial enterprise software increasingly being deployed on the cloud, it is important to adapt DevOps principles for assuring good QoS for cloud-based applications.

Multi-cloud platforms aggregates multiple cloud providers (for e.g., Amazon EC2, Microsoft Azure, Google Cloud Platforms, Private Clouds) to achieve faster, easier and better application building and deployment [3]. However, ensuring good QoS in multi-cloud platforms requires a substantial level of automation and adaptation in such heterogeneous hybrid platforms. This can be realized by employing a MAPE-K (Monitoring, Analysis, Planning, Execution-Knowledge) framework [4] [16][9], which deploys a feedback loop that cycles through Monitoring, Analysis, Planning, and Execution phases of a system, and shares data through a common Knowledge base. To support autonomic computing, there are existing enterprise products that follow the MAPE-K framework [10]. These autonomic frameworks are robust and reliable as they are designed to handle production applications. However, industrial MAPE-K frameworks support limited analysis based on basic threshold-based monitoring and planning [21]. This lack of flexibility limits the DevOps teams to create more sophisticated automation plans for complex multi-cloud applications. On the other hand, autonomous frameworks proposed in academia are more flexible and can be applied to a wider variety of analysis including machine learning models, analytical models, control loops, etc. However, they are less robust and non-scalable as many of them are not designed for production environments [24].

In this paper, we explore the feasibility of developing an autonomic MAPE-K framework for multi-cloud platforms by integrating existing services and Components-Off-the-Shelf (COTS) software. When we incorporate MAPE-K into applications to support DevOps operations, we need to satisfy several requirements. Multi-cloud application deployments must be dynamic in nature and require automatic scaling capabilities in which service components can be added or removed in response to incoming traffic without manual intervention. They also require adaptive management that enables deployment and movement of service components across different cloud providers. Providing good QoS in a multi-cloud platform needs large-scale automated resource monitoring across several cloud platforms, which is challenging to do since resource monitoring metrics are often tightly coupled with individual cloud providers. Additionally, the autonomic framework needs to be robust and should be able to handle failures as applications in production need to have high availability and stability. Essentially, we need to satisfy the requirements of availability, scalability, autonomic management, multi-cloud integration and ease of deployment. Keeping these requirements in mind, we answer the following research questions in this work:

- **RQ-1:** Is it feasible to build a MAPE-K framework for multi-cloud applications by integrating existing COTS and services?
- **RQ-2:** What are the challenges of building such a framework?
- **RQ-3:** What is the performance and efficiency of this framework?

RQ-1 explores COTS and available industrial services to determine if they meet our requirements. The interactions between each of technologies has to follow the MAPE-K model and needs to be connected together seamlessly. **RQ-2** identifies the various tasks needed for integration and the challenges we experienced when merging each of the services. **RQ-3** evaluates the framework on different self-adaptive use cases on a IBM benchmark application. We applied our framework to scale services depending on the workload, repair the application by redeploying a failed service and scale services in response to varying workloads on a multi-cloud setup.

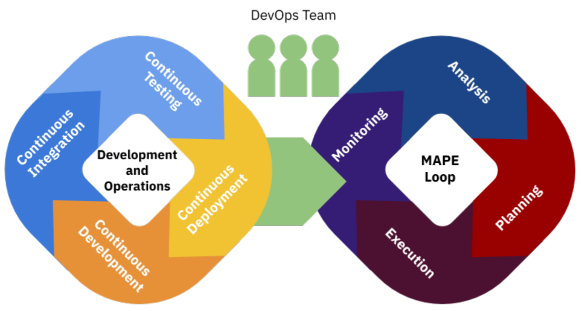

Figure 1: DevOps to MAPE Loop

We support the DevOps lifecycle by incorporating the MAPE-K framework with Continuous Deployment, which can be seen in Figure 1. Continuous deployment refers to automatically releasing the developer's changes of the application into production [1]. During application deployment, the DevOps team needs to monitor and manage their infrastructure to assure the services are properly provisioned and are able to perform well [2]. Our framework incorporates the COTS and available services to perform each of the phases in the MAPE-K loop. The DevOps teams can specify and develop the requirements that need to be satisfied for their managed application. The deployment and movement of the services on the cloud(s) can then be autonomously managed by our framework.

The rest of the paper is organized as follows. We discuss the background of DevOps and current research in self-adaptive systems in Section 2. We present our framework architecture and it's components in detail in Section 4. Section 5 outlines the concrete implementations of the framework components and our java code. We outline the setup of our testbed application for evaluation, and describe the use cases and the challenges that we faced when applying our framework in Section 6. Finally, section 7 evaluates the framework and presents experiment results from the use cases.

2 RELATED WORK

Autonomous Systems on Clouds has been a popular research area by practitioners and researchers, and integrating it for DevOps environments is a developing area. Cukier [11] present their experience in using cloud services to scale a web application by following DevOps and development patterns. They provide different solutions when scaling a complex web application that has a mix of cloud services in PaaS, SaaS and IaaS layer. Guerriero et. al. [13] designed a DevOps tool to support QoS assessment, optimization and QoS-aware runtime capacity allocation of cloud applications. To support DevOps operations with an Autonomous Management System (AMS), Barna et. al [7] proposed a method to develop an (AMS) for multi-tier, multi-layer data-intensive containerized applications. They employ a self-tuning performance model that outputs the performance metrics based on the application's topology, and plan for potential adaptive actions on the system if there are any problematic situations based on the performance model. In our research, we support the DevOps Operations by building an autonomic framework that explores COTS and existing tools to provide the DevOps teams with a high-level language equivalent to design adaptive actions for their system.

In Multi and Hybrid Cloud Autonomous Systems, different methods have been proposed to help scale deployments across multiple clouds. Wadia et. al [23] present a framework for elastic scaling of cloud resources that is portable across a wide range of private and public cloud providers and can be easily integrated with other frameworks. Their framework uses a Monitoring Engine, Decision Engine, Provisioning Manager, and a Database to auto-scale. Their decision engine is both rule-based and schedule-based, allowing them to scale at a particular time or when the monitored resource satisfies rules or policies. Miglierina et. al [19] propose a self-adaptive technique for both in-cloud scaling policies and traffic routing among the different cloud providers in a multi-cloud setup with a control-theoretical approach. Loreti and Ciampolini [18] propose an autonomic method to scale clusters of virtual machines over a hybrid cloud for MapReduce jobs. Kang et al. [15] propose an auto-scaling method to provide efficient resource utilization for Hybrid clouds. Their auto-scaler method is designed to handle variable workloads of modern applications, which must also meet SLAs such as deadlines, cost-oriented and performance-oriented policies. Ahn et. al design an auto-scaler that dynamically allocates resources depending on the two patterns of jobs, Bag-of-Tasks and workflow, in Hybrid Clouds [5]. Ahn and Kim also extend this work by focusing on dynamically allocating VMs in Hybrid Clouds in order to maximize resource utilization within a deadline and dealing with task dependency in workflow application [6]. Yunchun Li and Yumeng Xia [17] design a platform which can auto-scale web applications in hybrid cloud based on docker. They built a hybrid scheduling controller, and use a combination of prediction and reaction algorithms to scale docker containers of a web application on a hybrid cloud. In this paper, our architecture follows the MAPE-K framework, utilizing COTS and open-source services performing the Monitor, Analysis, Planning and Execution stages to autonomously manage Multi and Hybrid cloud applications.

Finite-state machine (FSM) is a design pattern that handles the change from one state to another in response to some inputs, based on a finite number of changes. Drumea and Popescu [12] discuss implementing finite state machines in the software application domain, with a focus on web technologies. Rules and the ruleflow can be modeled as finite-state machine with rules being used to transition from one state to the next [8]. Mor'n et. al [20] use a rule engine to apply adaptive changes in Federated clouds, with the adoption of Rule Interchange Format (RIF) to allow the portability of reusing the rules on a different rule engine. In our previous research, we used business rules to define security rules at an operational level and created a flow of rules to assist developers and security analysts in finding security vulnerabilities [22]. We extended this work by incorporating COTS and existing infrastructure management tools to ensure automated DevOps management for enterprise applications.

3 REQUIREMENTS FOR DEVOPS AUTONOMIC FRAMEWORK

In this section, we have established a set of requirements for a MAPE-K framework that automates cloud deployment and supports the DevOps process. These requirements were selected based on current challenges in Cloud Automation for DevOps operations and multi-cloud deployment for industry applications.

REQ-1: Availability Availability is a measure of how the system is able to run without failure and how fast it recovers from failure. Availability of a service is of primary importance in order to provide good QoS to its end users. Services deployed on a multi-cloud environment can often suffer from faults and must have mechanisms in place to recover quickly. This is challenging since a multi-cloud environment spans multiple cloud providers, with each provider offering different mechanisms of control and management over its own virtual infrastructure, which significantly increases the room for faults throughout multiple services and cloud platforms. Thus, we need highly reliable components that can be used by multi-cloud services in a production environment. This is most likely to be achieved by using established software components that have been already tested in production.

REQ-2: Multi-cloud Integration In a multi-cloud environment, service components are distributed across multiple cloud providers. In such environments, features from different public cloud provider platforms are required to be integrated with the service's internal private cloud. Since each cloud provider offers different mechanisms to control its infrastructure, integrating features and services as well as managing and automating service deployments becomes complicated in a multi-cloud environment. This involves runtime monitoring of data from multiple heterogeneous sources, deploying services on different cloud provider platforms and networking them with the proper security protocols in place.

REQ-3: Autonomous Configuration for Cloud Applications Services deployed on multi-cloud environments require an autonomic framework that is able to (a) deploy an application on multi-clouds and (b) self-manage the deployed application at runtime. To this end, the framework needs to handle a wide variety of input metrics from different service components deployed on multiple cloud platforms for sophisticated decision-making. Based on the decision, the framework should be able to perform a wide variety of different adaptive actions that can make changes to the service at runtime for enabling better service management and optimizing its QoS. Additionally, we also need to keep track of changes to our multi-cloud resources so that they can be used for implementing autonomic decision-making for managing our services in the future.

REQ-4: Scalability The autonomous framework needs to handle monitoring millions of metrics and deploying thousands of servers and applications across multiple data centers or cloud providers. When new services are deployed or new cloud platforms are integrated, the MAPE-K components needs to scale accordingly and seamlessly. We need to be able to scale up services at runtime in response to increased workload. Similarly, we should also be able to scale down services when needed so as not to incur additional cloud resource costs. This scaling has to be done automatically and efficiently so as to not react to transient changes in the state of the service.

REQ-5: Ease of Deployment We need to provide the DevOps teams an autonomous framework that can be integrated with the service without incurring significant levels of development, maintenance or learning effort. For this purpose, each of the COTS tools selected for the MAPE-K framework must be intuitive to learn, and easy to configure and implement. Selecting COTS tools with extensive documentation and online support will help significantly in this regard.

4 DEVOPS AUTONOMIC FRAMEWORK ARCHITECTURE

The architecture of our proposed framework as shown in Figure 2 has three main components: (1) Cloud Monitor, (2) State Rule Engine, and (3) Workflow Engine. These components interact with each other in a continuous MAPE-K loop to conduct DevOps automation and integration for cloud-based services. For the monitoring stage of MAPE-K, we deploy a Cloud Monitor that monitors and records performance metrics of the managed services running on different cloud platforms in a multi-cloud environment. The State Rule Engine is responsible for the analysis and planning stages of the MAPE-K loop where it queries events, i.e. a set of metrics from the Cloud Monitor, to automate operational decisions for changing the state of the deployed service, if needed. Finally, the execution stage of the MAPE-K loop is automated by implementing these operational changes by the Workflow Engine. We describe the Cloud Monitor, State Rule Engine and Workflow Engine components of our framework in details in Sections 4.1, 4.2 and 4.3 respectively. We also describe how our architecture adequately satisfies the 5 requirements (REQ-1 to REQ-5) as discussed before in Section 3.

4.1 Cloud Monitor

The Cloud Monitor is used to monitor and collect performance metrics for our managed service on the cloud. The Cloud Monitor

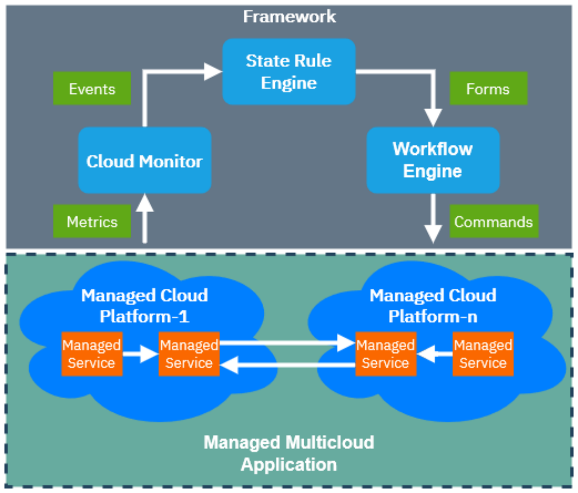

Figure 2: Conceptual Framework

deploys a distributed cloud monitoring tool on each cloud provider platform that runs the individual service components of the managed service. The Cloud Monitoring tool runs continuously along with the service to collect performance metrics at runtime. It can be configured to collect a wide variety of such metrics at both the application, platform and infrastructure level. In our case, we designed the Cloud Monitor to collect resource utilization metrics for each service component continuously at runtime at an interval of t seconds. We note that the value of t is configured by the DevOps team. This value needs to be tuned for each service component and cloud platform, depending on what kind of metrics the DevOps team wants to monitor as well as the performance overhead imposed by the Cloud Monitor on the service. In our case, we set the value of t based on the observation that collecting the resource utilization metrics from each service tier once every t seconds does not impose a maximum performance overhead of more than $2\%-3\%$. For multi-cloud setups, the cloud monitoring tool is distributed on all the cloud platforms, and the performance metric data from each cloud platform is streamed to the State Rule Engine and aggregated.

The Cloud Monitor uses *Service Discovery* to find new instances running the service for collecting the desired performance metrics from these instances. Service Discovery is a process for automatic detection of newly deployed services. In order to collect metrics from an instance, the metrics have to be exposed to the Cloud Monitor. This is done by using an *exporter.* An exporter exposes the performance metrics of the instance on which a service tier is running. The exporter is deployed on the same cloud instance as the service component. When an exporter is initialized, it has an exposed port which can be accessed by the Cloud Monitor to capture the metrics from the instance. When a new instance is deployed along with its exporter, the Cloud Monitor uses Service Discovery to find the instance first. We note that the Cloud Monitor component in our framework can also be used to work with the microservice architecture. To this end, the exporter is deployed as a container on each cloud instance running the service. We note that for containers, the container technology itself exposes the performance metrics of its containers, rather than each individual container. Thus, Service Discovery is not required for new containers. For multi-cloud setups, each of the instances and container platform on the different cloud platforms have their own exporter

The ability to deploy the Cloud Monitor on multiple cloud platforms allows the DevOps team to enable multi-cloud integration (REQ-2). The Cloud Monitor can automatically handle a large selection of input metrics from heterogeneous cloud platforms, which fulfills the requirements of autonomic management (REQ-3) and scalability (REQ-4). Finally, the DevOps team can easily deploy the Cloud Monitor and integrate it with the rest of the components in our framework (REQ-5). The Cloud Monitor provides an API for the other components of our framework to access. The State Rule Engine periodically queries the Cloud Monitor API. The query can include functions, for e.g. sum and average of all metrics collected over the past T seconds. The aggregated data is then sent to the State Rule Engine as an event for the Analysis and Planning stage, as described in detail in Section 4.2 next.

4.2 State Rule Engine

The State Rule Engine component is responsible for the Analysis and Planning part of the MAPE-K loop to automate the deployment of the services. The State Rule Engine is used to execute and manage operational rules. These rules are composed of a set of conditional and consequential "When - Then" rules, for e.g., *When* a condition occurs, *Then* perform a consequence or action. Since rules are simple to learn and can be easily understood by the DevOps teams (REQ-5), we implement the State Rule Engine to facilitate autonomic decision-making. The rules are written by both operations and development teams based on the events that are input from the Cloud Monitor to the State Rule Engine. The State Rule Engine enables the DevOps teams to collaborate for maintaining a desired level of QoS for the deployed service at runtime in a large multi-cloud environment. The operations team can utilize the rules to manage and deploy the required infrastructure and platform for the services, while the development team can autonomously configure their application as services on the infrastructure at run-time through the rules. As mentioned earlier, the State Rule Engine queries events from the Cloud Monitor periodically every T seconds. These events are then validated against the conditionals of the rules, where first the "When" statement is checked to be satisfied against the current values of the events. If the conditional is met, the consequential statement is triggered, which performs the action from the "Then" statement mentioned in the rules.

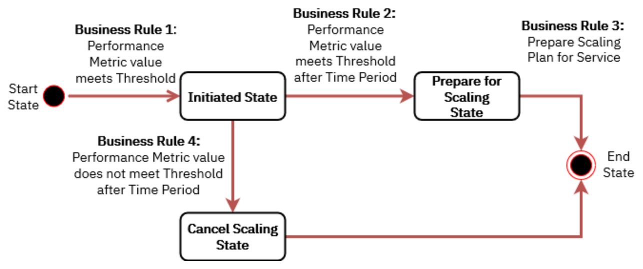
Business Rule 1: Performance Metric value meets Threshold
Business Rule 2: Performance Metric value meets Threshold after Time Period
Business Rule 3: Prepare Scaling Plan for Service
Start State
Initiated State
Prepare for Scaling State
Business Rule 4: Performance Metric value does not meet Threshold after Time Period
End State
Cancel Scaling State

Figure 3: Analysis Stage of State Rule Engine

The consequential ”then” statement can perform the Analysis stage or Planning stage. We can take advantage of the stateful and temporal properties of the State Rule Engine. The State Rule Engine can store objects in memory. These objects include states, metrics and deployment descriptions. Since these objects are stored in memory, we can perform analysis by having rules that compare new objects with the previous objects that are stored in memory. The temporal property means that the objects can have a timestamp, which allows for time-based conditionals in the rules.

For the Analysis stage, we use states in the State Rule Engine that are used to define the current event of the service. The states take advantage of the temporal property which allows us to compare the timestamps of the state and another object. When a business rule is triggered, it can begin the Analysis stage by inserting a new state. An example of this can be seen in Figure 3, where in the beginning Business Rule 1 compares the performance metric event against a threshold value. If the metric meets or exceeds its threshold value, the state of the service is transitioned to an initiated state and an adaptive action needs to be taken, for e.g., scaling up the cloud service. The Business Rule 2 is triggered after another sampling interval and uses the temporal property to compare the current performance metric event with the initiated state to check if the metric still meets or exceeds the threshold value. If so, the business rule causes the state to transition to the next state. This triggers the Business Rule 3 in the sequence and starts the Planning stage which prepares the service for scaling. However, if the performance metric no longer meets the threshold value, Business Rule 3 is triggered that cancels the state from the State Rule Engine, as shown in the figure. The use of states in the State Rule Engine shows forward chaining in action, where the rules are connected with each other in sequence.

For the Planning stage, service deployment and modification plans can be defined in the consequential "Then" statement. Plans are the deployment details of the service. Examples of these deployment details include the number of container replicas, hostname, cloud platform location, etc. The cloud operator can specify these input values of the instance that will need to be deployed or modified. The new values will be sent as a POST request to the Workflow Engine's API, where the changes will be applied. The Workflow Engine will then return a response, which gives the details of the service. These details include the name and ID of the deployment, the output value of the services (e.g. IP address) and the input values that were specified by the rule. We then store these details into the State Rule Engine, which can be used for future conditionals and consequential. For example, if we need to deploy a second service that requires the previous service, we can create a conditional statement that will trigger if first service has already been deployed by checking if the deployment details are in the State Rule Engine.

Using the State Rule Engine allows the DevOps team the robustness (REQ-1) to maintain QoS in large services spanning multiple cloud platforms by taking adaptive actions through a ruleset. Using the State Rule Engine, we can orchestrate the scaling of multiple services (REQ-4) deployed on different cloud platforms with rules that are easy to understand. This means integration in the DevOps process will be more efficient with minimal impact on the actual application which improves the ease of deployment (REQ-5).

4.3 Workflow Engine

The Workflow Engine is responsible for deploying and managing the services on the cloud through end-to-end automation. It is responsible for the execution stage of MAPE-K by deploying the changes based on the analysis and planning decisions from the State Rule Engine. The Workflow Engine uses Templates, which is a blueprint of the service and infrastructure details described in high-level configuration syntax. We use Templates to describe the instances and configurations required for running the service. The services are then deployed by the commands issued by the Workflow Engine based on the Template descriptions. Additionally, the Workflow Engine is also responsible for automating the initialization and runtime of the service. Once the service is deployed, the Workflow Engine can modify and apply changes to the running instances, for e.g. applying new updates to the service, making changes in the requirement or taking auto-scaling decisions. Hence, we can automate DevOps operation by integrating the convenience of rules along with the Infrastructure-as-Code (IaC) capability of the Workflow Engine.

The Templates are written in IaC scripts, and the IaC engine is used to deploy and manage these Templates. The deployment variables, such as the access keys, number of VMs, resource allocation, etc, are specified by the DevOps teams and are obtained as input from the State Rule Engine component. The Template can select one or multiple cloud providers where the services will be deployed, thus automating service deployments in a multi-cloud environment (REQ-2). As Templates are high level descriptions of our services, it is easy to keep track of a large number of growing services, supporting the scalability of those services (REQ-4). Using IaC scripts in the Workflow Engine also provides more flexibility for the DevOps teams to describe, modify and manage the service configurations at runtime with ease (REQ-5). We note that this is better than designing, building, testing and maintaining an adaptive framework from scratch, since each of the components has already been extensively tested and designed for production environments. Additionally, the industrial tools used for building the framework components have the added advantage of good documentation and tool support.

RQ-1: It is feasible to develop MAPE-K frameworks for multi-cloud applications by reusing COTS. There is an abundance of commercial and open source services, especially for implementing the Monitoring and Execution components. The offering for Analysis and Planning is weaker, which suggests that more research is needed in developing reusable services for these two activities. A mitigation strategy is to use state-full rule engines, which can encompass both Monitoring and Analysis. Rule engines, given their low complexity but high expressiveness, can also act as a language common denominator for a Development and Operations team.

5 CONCRETE ARCHITECTURE

In this section, we present the COTS that we selected to build our MAPE-K framework and describe the integration process for each of the COTS. Table 1 show the COTs that we used for each component, as well as other potential alternatives COTS.

Component	COTS
Cloud Monitor	Prometheus, Graphite, InfluxDB, Nagios, Zabbix
Rule Engine	Drools, IBM Operational Decision Manager, Red Hat Decision Manager, Grule
Automation Manager	IBM Cloud Automation Manager, Ansible, AWS CloudFormation
Container Platform	Docker, Kubernetes
VM Exporter	NodeExporter
Container Exporter	cAdvsior

Table 1: COTS for Each Component

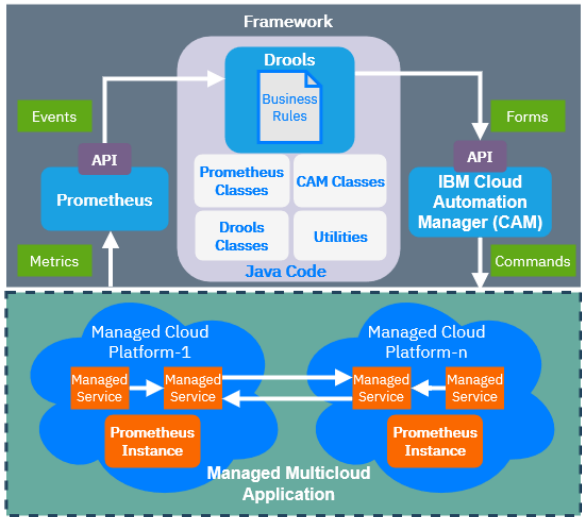

Figure 4: Concrete Architecture

5.1 Prometheus

For our Cloud Monitor implementation, we used Prometheus [1], an open-source monitoring system and time-series database that can collect runtime performance metrics at runtime. Prometheus stores runtime performance metrics collected from an exporter into its local memory and allows us to query the stored metrics. We can use query functions provided by Prometheus to get the aggregate time series data in real-time.

Prometheus integrates data exporters, including third party data exporters, depending on the cloud platform where the service will reside on. For capturing the performance metrics of containers, we use a exporter from Google called cAdvisor [2] which provides the resource usage of running containers. For Virtual Machine, we use Prometheus' NodeExporter to capture the performance metrics of the hardware and the Operating System. In our implementations, we deploy a cAdvisor container on each of the Docker Nodes on our Docker Swarm Cluster. On the Docker Nodes, the cAdvsior exposes its metrics on a specified port (the default port is 8000). In the Prometheus configuration file, we define the cAdvisor port for Prometheus. When new containers are deployed or old containers are deleted, the performance metrics of these will automatically be reflected on Prometheus. For our Hybrid Cloud implementation, we use both cAdvisor on an Openstack-based community cloud, which will be described in more detail in Section 6.1.4, and NodeExporter on the EC2 Cloud. When we deploy a MongoDB instance on EC2 through the IaS script, we also specify in the Terraform script to install and enable NodeExporter on the MongoDB EC2 instance. In the Prometheus configuration, we enable Service Discovery to filter VMs with the String "MongoDB" in its name. NodeExporter metrics are exposed on port 9100 by default, which we specify in the Prometheus Configuration for Service Discovery. For both Node Exporter and cAdvisor, we have to specify the time interval in which we want to collect the metrics in the Prometheus configuration. For our experiments, we collected metrics of cAdvisor and NodeExporter every 5 seconds. The default retention time of the data is configured as 15 days.

[1]https://prometheus.io/
[2]https://github.com/google/cadvisor

5.2 Drools

For State Rule Engine implementation, we use Drools [3], an open-source Business Rules Management System (BRMS). Drools is implemented and located on-premise in the IBM Cloud. Drools is Java-based, and it is implemented with our own Java code. This is through Java Libraries provided by Drools that contain the Drools core and compiler. We use the classes provided by the Drools Libraries in our Java Code which will be explained in more detail in Section 5.4. The Drools Engine handles a Drools file that contains a set of Business Rules. To get the metrics from the performance monitor, we have an initial rule that calls the Prometheus API with a query range that gets the utilization metrics from the last 30 seconds. The rule also uses one of our custom Java methods that calculates the average CPU utilization from the query range of values. To insert the metric into the memory, we create a Drools Type Declaration. While Drools works out of the box with plain Java Objects as facts, the user may want to define a fact directly in the Drools File instead of creating a new object in Java. Drools provides the ability to declare new types in the Drools File to be used with the business rules. In our use cases, we defined a new type called Metrics with the attribute cpuAverage to store the CPU Average from the query range and insert it into the Working Memory.

The auto-scaling decisions in the Business Rules are executed using States and rule chaining. The first rule evaluates the performance metrics against a threshold value in the rule conditional. If the rule is triggered, a state is created and inserted into the working memory. In the state, we set the name and state value. In our use cases, the state value is binary, either the state is new and has not completed the task, or the state has completed the task. Using states, we can create a sequence of rules that react to each others changes, known as Forward Chaining. Instead of reacting instantly to the change in the performance metrics, we insert the state for the next rule to evaluate. If the performance metric is still within the threshold value of the condition and the state has been in the working memory for 30 seconds, the next rule in the chain is triggered. This rule sets the state as FINISHED for the third rule in the sequence.

[3]https://www.drools.org/

When the state is set as FINISHED, the rule is triggered and begins to call the Automation Manager API and scale the services accordingly. The Drool Rules are able to call the CAM API through our custom Java class to send the deployment and modification details for the services.

5.3 CAM

For our Workflow Engine, we use IBM's Cloud Automation Manager (CAM) [4] to handle the deployment of services. It uses the Terraform Engine [5] to deploy services that are described in IaC scripts. Using IBM CAM strengthens the robustness in our autonomic framework as it is an industry component rigorously tested and deployed on actual production environments. CAM is deployed on an internal IBM VMware Cloud. We first need to initialize a cloud connection to our Cloud Platforms. We set the authentication values of our Openstack-based and EC2 cloud connections. For SAVI, we require the Project Name, Domain Name, Region Name and Login credentials. For connecting to EC2, we need the Access Key ID and Secret access key. We have to create a template for our services that we plan on deploying on the cloud platforms. In practice, the Template and its associated files is located in a GitHub repository. When a new commit is placed on the repository, the changes are reflected dynamically in the CAM Catalog.

```
resource "docker_service" "web-application" {
  name = "${var.webname}"
  mode {
    replicated {
        replicas = "${var.web_replica}"
    }
  }
    task_spec {
        container_spec {
            image = "webapp/image-name"
            env {
                MONGO_PORT_27017_TCP_PORT="${var.mongodbport}"
            }
        }
    }
}
```

Figure 5: CAM Template of Web Application

Our services and their configurations are described in the Terraform template. An example of a portion of our service Template for a web application can be seen in figure 5. This web application resource is a deployment for Docker Swarm. As seen in the figure, the var.variablename in the template are the input deployment variables. The number of replicas is one of the variables that we modify with our Drools engine for scaling. Other configuration values that can be seen are the name of the web application image and the environment variables in the *env* block. When the CAM API receives the POST request to plan, modify and apply the Template, it begins the process of deploying the instance on the cloud. Depending on the service, it may take a few or several minutes for the deployment to be complete. Once complete, CAM displays the output variables automatically, such as the IP address.

[4]https://www.ibm.com/us-en/marketplace/cognitive-automation
[5]https://www.terraform.io/

5.4 Java Components

To be able to get Drools functioning with Prometheus and CAM, we had to code and implement our own components to support the framework. Since Drools is Java-based, all our components were coded in Java. As seen in Figure 4, there are four main components: (1) Drools Classes (2) Prometheus Classes (3) CAM classes and the (4) Utilities. The Java libraries provided by Drools are imported in a Core java component that we also coded. This core component runs the Drool Engine and orchestrates between all the other components. These Java components are used to connect Prometheus, the Drools Engine and CAM in a working MAPE-K process.

The core components build the Drools Engine and input the Drool rules file into the engine. Building the Drools engine with our Java code first requires creating a Drools memory file system which is provided by the Drools library. We then import our Drools rule file location into the Drools memory file system. A Drools configuration class is also declared to specify the STREAM option which adds a duration attributes to the rules and memory management techniques to optimize real-time streaming. Finally, we initiate a Drools session from the Drools file system detailed above. The session is used to fire the rules periodically every 30 seconds. The core component also periodically checks if there is any change in the rule files. When there is a change in the Drools file, it rebuilds the session with the modified Drools file.

RQ-2: The main challenge in developing a MAPE-K from COTS is the development effort for integrating components. While the integration of the Monitoring and Execution components with the underling cloud and the application is of medium complexity, integrating the Rule Engine with Monitoring Execution requires some effort. For our particular case, we needed to write and test 1760 lines of code. A second challenge is the steep learning curve of particular components as well as of the underlying cloud infrastructure.

6 EXPERIMENTS AND EVALUATION

In this section, we describe the testbed for our evaluations and how we applied our MAPE-K framework to three self-adaptive use cases.

6.1 Testbed Applications

6.1.1 Acme-Air. In our work, we use a Web benchmark application called Acme-Air [6] developed by IBM. Acme-Air is an implementation of multi-tier airline e-commerce microservice application composed of two service components. The application-tier component is a front-end NodeJS server connected to a data-tier component, a MongoDB Database. These components are deployed as Docker containers that can be run on different cloud platforms. We selected the micro-service application mode and containerized the Acme-Air components by enabling a Docker Swarm Cluster. This allows us to to easily scale the Acme-Air web and database servers using our framework. We used the httperf and JMeter workload generator tools to generate traffic to our Acme-Air application which allows

[6]https://github.com/acmeair/acmeair

us to control and stress the system metrics (CPU, memory, disk, network) for our use cases.

6.1.2 HAProxy Loadbalancer. We implemented a containerized HAproxy loadbalancer [7] for our Acme-Air web containers. We deployed the HAProxy container on the Docker Swarm Cluster where the Acme-Air services reside. We used the containerized HAProxy loadbalancer to distribute the workload to the Acme-Air containers within the cluster. We enabled service discovery in the HAproxy configuration file by looking for newly deployed AcmeAir containers by their container name and connecting them to the loadbalancer. In our use cases, we instruct HAProxy to look for container services that has "Acme-Web" as their service name. When a new Acme-Web container is deployed, HAproxy looks for that container that matches the name and port number that is specified in the server-template setting. The workload generator sends the requests to HAproxy loadbalancer as the entry point to Acme-Web containers and we used the default round-robin load balancing algorithm.

6.1.3 Docker Swarm. The Acme Air Service components service components were containerized to be deployed on a Docker Swarm Cluster. We set up a Docker swarm cluster with three nodes for our deployment. All the nodes were on medium-sized VMs (2 VCPUs, 4GB Ram, and 40GB disk). Two of the nodes were worker nodes while one of the nodes was both the master and worker. This allowed us to run Acme-Web in a distributed environment with a large amount of CPU and disk resources to stress using the workload generator. Since the web service and the mongoDB containers connect with each other from different nodes, we create a Docker overlay network. An overlay network sits on top of the host-specific networks allowing all containers in the network to communicate. The HAProxy loadbalancer is deployed on the master node which has a public IP address and acts as the entry-point. All the other worker nodes are internal and only have private IP addresses.

6.1.4 Cloud Platform. We deployed our services on SAVI Cloud [8] and EC2 Cloud[9], while the BRE and Automation Manager were deployed on premise in an internal VMWare Cloud [10]. SAVI is a community cloud provided as a partnership between universities, research and industry in Canada. It is built on Openstack [11], which is one of the many clouds supported by the services we used to build our framework. For our Hybrid Cloud use case, we deploy the Acme-Air database services on EC2. Since Workflow Engine will need to use the API of both SAVI and EC2, the authentication values are required by the Workflow Engine. For Openstack, these values can be found in the Openstack RC File V3. The Authentication URL, Region Name, Project Name, Domain Name, Username and Password are required for the Workflow Engine to connect to the Openstack API. For EC2, the Access Key ID, Secret Access Key and Availability Zones are required for the Workflow Engine. Since the State Rule Engine and Workflow Engine are located on the IBM internal cloud, these components also need to also connect to the performance monitor and the Docker Swarm API to scale the containers. Public floating IP is associated for the Docker Swarm Master node and the Cloud Monitor instance. We use security groups for both these instances, and create rules such that only IP addresses of State Rule Engine and Workflow Engine can access the exposed ports of the Cloud Monitor and Docker Swarm APIs.

6.2 Use Cases

6.2.1 Self-Configuration. It is important for a web application to prevent the end user from experiencing high latency and availability issues. To this end, web applications should be able to self-configure automatically for providing good quality of service while optimizing costs incurred due to resource utilization. This allows for the support of Continuous Deployment by keeping the application within good performance requirements. We demonstrate the self-configuration capability of our framework in the first use case using the Acme-Air Web application. Our Acme-Web and MongoDB containers are all deployed on the three-node Docker Swarm Cluster. We test our framework's self-configuration ability to see if it can keep the average CPU utilization of the Acme-Web and MongoDB containers within an acceptable threshold range. For this purpose, we set the upper and lower threshold limits for the average CPU utilization in Acme-Web to 50% and 25% respectively. For our MongoDB container, we set the upper and lower threshold limits for the average CPU utilization to 10% and 5% respectively. We first deploy our performance monitor and initiate service discovery, as mentioned in Section 5. Next, we write business rules in Drools that can handle the self-configuration scenario.

The first rule we create is to insert the Prometheus metric into the Drools engine. This rule triggers periodically after an interval set up by the DevOps team. In our case, this was done by setting the timer attribute for the rule to 30 seconds. In the "Then" statement, we get the query range from the last 30 seconds of the average CPU utilization from our Prometheus component, which we insert into a Metric object and store it in the Drools memory, as mentioned in Section 5.2. The first business rule for self-configuration checks if the Acme-Web containers have an average CPU utilization over 40% and whether there is any existing state in the memory for scaling. When the threshold and requirements of the "When" statement in figure 6 are met, we initialize the state. As seen in the figure, we give the state an identifying name "ScaleUp" to allow the other rules responsible for scaling up to look for this state. We set the state value to NOTRUN, which is the initial value of the state. We then insert this state into the working memory. The next rule checks if the "ScaleUp" state exists in the working memory, and if it is in the initial state. This rule executes if the average CPU utilization is still over 50% after 30 seconds of the initial state. The state is set to FINISHED for the final rule in the sequence to trigger. The reason why we have these two rules is to check if the average CPU usage is consistently above the upper threshold limit of 50% since we want to prevent reacting to sudden and transient utilization spikes.

The rule that is responsible for the execution stage and sending the scaling plan to CAM is shown in Figure 7. This rule checks if the State is complete and inserts a camJson() object as seen in the figure. This camJson object includes deployment value of the number of current Acme-web replicas. We initialize the CamTemplateAPI from the Cam Caller Component in the "Then" statement, specifying the endpoint of our CAM deployment. We get the current value of the

[7]https://hub.docker.com/r/haproxytech/haproxy-debian
[8]https://www.savinetwork.ca/
[9]https://aws.amazon.com/ec2/
[10]https://cloud.vmware.com/
[11]https://www.openstack.org/

```
rule "Prepare for Scale Up"
  when
    metric: Metric (metric.getCpuAverage() > 50)
    state : State (name == "ScaleUp" && state ==
         State.NOTRUN, this before[30s] metric)
  then
    state.setState(State.FINISHED);
    update(state);
end
```

Figure 6: Business Rule for Finishing State

```
rule "Scale Up" salience 10
  when
    camJson : CamJson ()
    state : State (name == "ScaleUp" && state ==
         State.FINISHED)
  then
    CamTemplateAPI camAPI = new
         CamTemplateAPI("<HOSTNAME>");
    int value = camJson.getNumericalValue("acme.json",
         "web_replica");
    camJson.changeValue("acme.json", "web_replica", value
         + 1);

    CamJson ModifyJson = new CamJson();
    camAPI.ModifyInstance("<instance-id>","acme.json")
    camAPI.ApplyInstance("<instance-id>")
    retract(state)
end
```

Figure 7: Business Rule for calling CAM to Scale

replicas for our Acme-Web Service using the getNumericalValue() function, which is a simple function that parses the Json of the Acme-Web deployment. We increase the value of the replica in the Json file. We send the Json file through a POST request for modifying an instance to the CAM API and then send a POST request for applying the modification on the SAVI cloud. We then retract the state, allowing the process to accept new "ScaleUp" states. This process is same for scaling down as well. If the average CPU usage does not meet the threshold value while a "ScaleUp" or "ScaleDown" state is already in the memory after 30 seconds, another rule is triggered that removes the state, and restarts the rule sequence.

There were challenges when setting up our Self-configuration use case with our framework. When we first began to deploy the Acme-web infrastructure through Terraform, both our MongoDB and Acme-Web containers resided in a single Template. However, due to Terraform's behavior, this caused some issues. As mentioned before, the Automation Manager may need to recreate the instance when a Modify plan is applied. When having Acme-Web and MongoDB services in a single template file, we observed that when scaling the Acme-Web containers, it would remove the MongoDB container each time. We decided to create two different templates, one for MongoDB and another for Acme-Web. This would allow us to scale without causing an increased unavailability time by removing the MongoDB container. We also needed to figure out a way to keep track of the number of replicas when we scale. Originally, we started our BRE with no services deployed on SAVI and wrote rules that deployed the initial infrastructure before scaling. However, most of the time there is already an existing service on CAM that we may want to modify. For this reason, we choose to deploy and modify services by using Json files. With the acme.json file, we can keep track of the web replicas of an existing service. However, one related challenge that is unresolved is tracking changes to services that occur through the CAM UI with our BRE. If the DevOps team makes changes to the services through the CAM UI, we need to be able to sync those changes while the BRE is running.

6.2.2 Self-Healing. When a service goes down, we need the ability to automatically respond and re-deploy the service. We demonstrate our framework's capability to self-heal a service in the second use case. This can support the DevOps team and Continuous Deployment by maintaining good availability while potentially logging and alerting the DevOps team when the service went down. To this end, we use our framework for self-healing Acme-Air by automating the redeployment of a MongoDB container when it goes down instead of redeploying MongoDB manually. We check the status of MongoDB for Acme-Web to see if we get a response. This is done through our Prometheus instance, which is periodically monitoring the MongoDB instance. If we receive an empty or irregular JSON array from Prometheus, we conclude that the MongoDB service is down. Instead of inserting the CPU usage average metric into the working memory, we insert the JSON output when we query Prometheus every 30 seconds.

```
rule "Prepare for Self-Healing"
  when
    jObj : JsonObject
         (jObj.get("data").getAsJsonObject().get("result")
         .getAsJsonArray().size() == 0)
    state : State (name == "HealDB" && state ==
         State.NOTRUN, this before[30s] jObj)
  then
    state.setState(State.FINISHED);
    retract(jObj);
    update(state);
end
```

Figure 8: Business Rule for Self-Heal State Initialization

Similar to the rule that initializes the state in the Self-Configuration use case, we run a rule that checks if the query of our MongoDB service is empty. If empty, the rule inserts a new state named "HealDB". The next rule in this sequence is shown in Figure 8. The "When" statement checks if the query is still empty after 30 seconds of the initial state. The rule is then triggered, and the state is complete. The time duration between the two rules that modify the state is to confirm that the MongoDB service stays down consistently. The final rule in the sequence checks if the state is complete, and uses the ModifyInstance and ApplyInstance functions seen in Figure 7 to MongoDB.

6.2.3 Hybrid Cloud Scaling. One of the key features of CAM is that it allows the DevOps team to automatically deploy services in a Hybrid cloud setup. An organization can have private data that they prefer not to store on a public cloud and store it on their internal private cloud instead. There also may be unique services that are only offered by a specific cloud platform and the organization may want to use different services from different cloud providers simultaneously, such as using storage instances from one cloud platform while having front-end services on another cloud platform. Resource and cost constraints can also be a reason to have a Hybrid Cloud setup. Our framework enables the capability of CAM to allow for automatically managing services in a Hybrid Cloud setup. In this use case, we scale a MongoDB instance in EC2, while simultaneously scaling an Acme-Web container in our SAVI cloud. Similar to the Self-Configuration and Self-Healing use case, we have a rule that checks the CPU average usage of the MongoDB instance deployed on EC2. We implement business rules for scaling up and scaling down these instances in a Hybrid Cloud setup. Since the Acme-Web container needs to connect to a running MongoDB, we first have to deploy a MongoDB instance. When the state is finished after the 30 second timer, the business rule sends a POST request to CAM to deploy a new MongoDB instance on EC2 with a json file. The json file contains the deployment variables for our MongoDB instance. Once we deploy the MongoDB instance on EC2, we can then deploy an Acme-Web container in SAVI that connects to MongoDB through the next business rule in sequence. We write our rules such that they prevent the Acme-Web container to be deployed while the Mongo-DB deployment is still not complete.

When scaling instances on a Hybrid Cloud platform, we had to consider some unique challenges. Since we have our services on two different cloud providers, we have to deploy two cloud monitors, one on SAVI and the other on EC2. Every time we have to deploy a service on a new Cloud Platform, we have to deploy Prometheus with its respective Service Discovery configuration as it can differ for each cloud platform. Next, we needed to automate the connection of the MongoDB instance on EC2 to the Acme-Web containers in SAVI with our framework. By deploying MongoDB as an instance, CAM is able to get deployment output variables from the EC2 instance, and in our case, the public IP address of the MongoDB instance from EC2. We can then specify in our rule that the Acme-Web container will connect to the MongoDB instance with that IP address. This simplifies the automation process to establish the connections between the two services by using CAM to get the output variables of newly deployed services. Since we have our Database on EC2 and our Web Server on our SAVI cloud, we need to automate ways to secure the connection between Acme-Web and MongoDB. We first have to manually create a security group that only exposes the MongoDB port to the IP address of SAVI Instances used for outbound requests. With CAM, we can specify the unique ID of the security group in the template which it will automatically associate the MongoDB instance with our security group during deployment.

7 RESULTS AND DISCUSSION

The results of the Self-Configuration use case are shown in Figure 9 and Figure 10. We start with one Acme-Web container and gradually

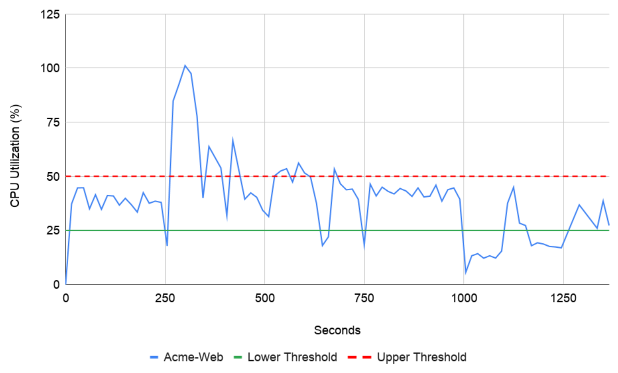

Figure 9: Acme-Web CPU Utilization during Self-Configuration

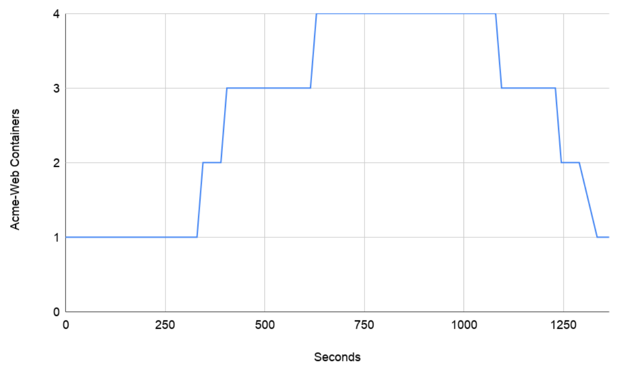

Figure 10: Number of Acme-Web Containers during Self-Configuration

increase the incoming workload which causes the CPU utilization of the Acme-Web container to exceed its upper threshold limit, as seen around the 270 second timestamp in Figure 9. Our framework correctly identifies this transgression, and triggers the business rule that waits for an additional sampling time of 30 seconds before scaling. This rule automatically adds a second Acme-Web container to the service at the 345 second timestamp as seen in Figure 10, taking 35 seconds to deploy the container from when the rule to scale up is first triggered. The average CPU utilization is still above the threshold at this point, and the framework adds an additional container at the 405 second timestamp. Our framework successfully self-configures by distributing the incoming workload between the 3 containers such that the average CPU utilization of the Acme-Web containers is maintained within the acceptable threshold limits for 75 seconds until the workload is increased again. To this end, our framework adds a total of 4 Acme-Web containers to the service and the average CPU utilization stays inside its acceptable threshold values from the 630 second timestamp to the 1005 second timestamp as we no longer increase the workload. At this point, the workload generator starts decreasing the workload to the service, which triggers the scale-down business rule in our framework. The utilization decreases below the lower threshold at the 1020 second timestamp, after which begins the additional 30 second sampling period before scaling down. This rule takes 35 seconds to remove the Acme-Web container. However, even after removing a container, the CPU utilization stayed below the lower threshold limit. Our framework successfully identifies this transgression and further removes two containers, the first container at the 1245 second timestamp and second container at 1335 second, thus maintaining the average CPU utilization within the threshold value.

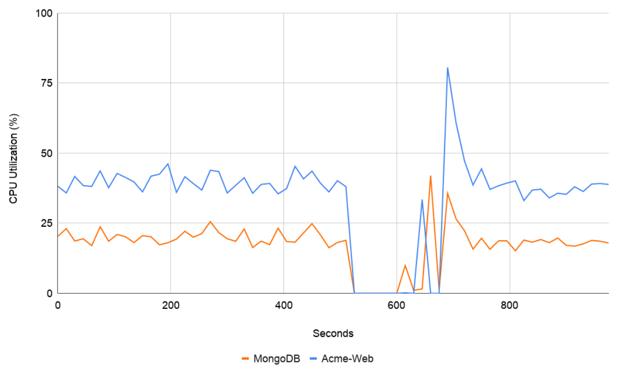

Figure 11: Acme Air CPU Utilization during Self-Healing

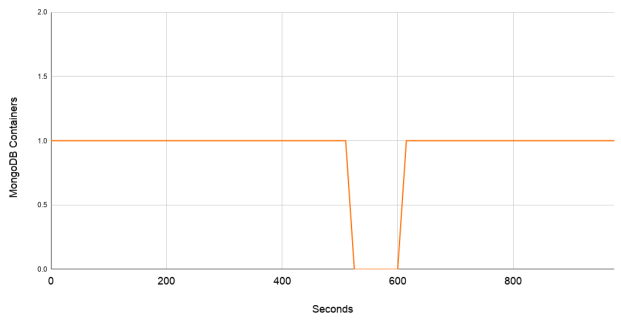

Figure 12: Number of MongoDB Containers during Self-Healing

Next, we discuss results for the Self-Healing use case as shown in Figure 11 and Figure 12. We configure the Acme-Air service with one Acme-Web and one MongoDB container and send incoming workload to the service such that the CPU utilization values for both the containers are at acceptable levels. At the 525 second timestamp, the MongoDB container goes down, which reduces the CPU utilization values for both the MongoDB and Acme-Web containers to zero, as seen in Figure 11. This indicates that at this point, the MongoDB container is not working and the Acme-Web container, which is dependent on the MongoDB, also stops working. Our framework correctly identifies this fault after waiting an additional 30 seconds and triggers the self-healing business rule described previously in Figure 8 to correct this fault. Consequently, the framework automatically re-deploys the MongoDB container, as seen in Figure 12. which takes 42 seconds. Once the container is deployed and running, it takes an additional 90 seconds for the MongoDB container to stabilize, after which the container utilization levels are within acceptable limits.

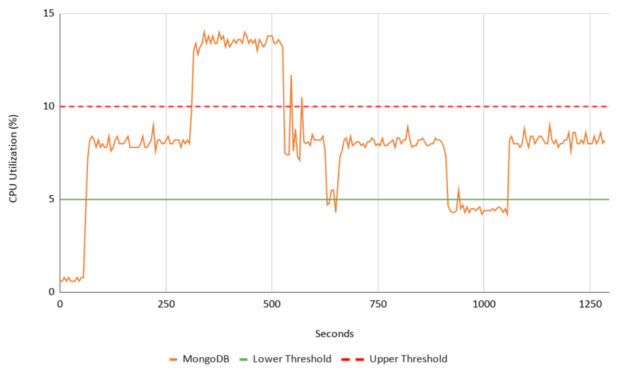

Figure 13: MongoDB Average CPU Utilization on EC2

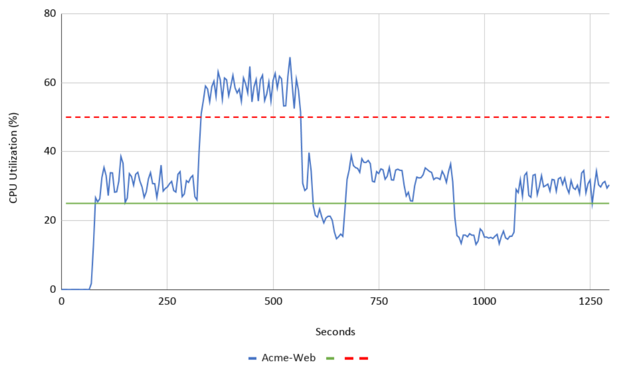

Figure 14: Acme-Web Average CPU Utilization on SAVI

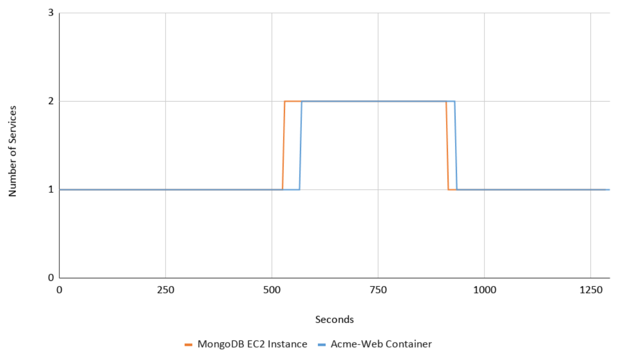

Figure 15: Number of Acme-Web and MongoDB Containers

Finally, we show results for the Hybrid Cloud Scaling use case in Figures 13, 14 and 15. We deploy this scenario using one Acme-Web container on the private SAVI cloud platform and one MongoDB instance on the public EC2 cloud platform. We set the upper and lower average CPU utilization thresholds for MongoDB to be within 5% to 10 % CPU Utilization since MongoDB on EC2 incurs lower levels of CPU Utilization than Acme-Web on SAVI. We start by sending incoming workload to the Acme-Air application such that the CPU utilization of both Acme-Web and MongoDB are maintained within the acceptable threshold limits. We then increase the workload so that their CPU utilization values exceed their upper threshold limits at the 325 second timestamp. Our framework correctly senses the need for scaling the hybrid cloud setup and triggers the corresponding business rules. The rules then scales the hybrid cloud automatically by adding a MongoDB instance in EC2 and then an Acme-Web container in SAVI, as seen in Figure 10. Since we are deploying MongoDB as an EC2 instance instead of a container, it takes longer to deploy because it needs to initiate and install MongoDB. The framework then evenly distributes the incoming workload between the 2 Acme-Web containers such that the CPU utilization values of Acme-Web and MongoDB are maintained within their respective threshold limits, as seen from the figures. Finally, we reduce the workload to the application at 925 seconds, which triggers a business rule for scaling down the new deployments. The rule deletes the newly deployed Acme-Web container from SAVI and the MongoDB instance from EC2 at 1070 second timestamp and is able to maintain the average CPU utilization within acceptable limits.

RQ-3: The MAPE-K framework we proposed, has the required performance and efficiency. We showed that it supports deployment and self-configuration in a multi-cloud environment, one of the main tasks of the DevOps pipeline. Furthermore, it can implement self-configuration and self-optimization across multiple clouds. Similarly, it can implement self-healing scenarios. The syntax of rules, used to implement the self-* is accessible to both Development and Operations team, making it appropriate for DevOps.

8 CONCLUSIONS

Enabling automation for service deployment and configuration in the cloud for the DevOps practice allows us to maintain good Quality-of-Service and continuous delivery. In this work, we presented a industrial framework that can support the automation of services on the cloud. We detailed the three main components of our framework, the Cloud Monitor, State Rule Engine and the Workflow Engine. These components interact with each other in our framework implementation based on the MAPE-K feedback loop.

In the future, we plan on expanding our framework with support from predictive models. Proactive self-adaptation methods can help avoiding costs incurred from excessive scaling as found in reactive approaches. This allows us to save costs and have better accuracy in various adaptation scenarios. We plan on implementing a predictive Machine Learning/AI model as a service that can be accessed by our framework components. Additionally, we plan to study our framework on different types of applications and services. We also plan on investigating our framework for more Hybrid Clouds scenarios as various large industry partners are moving towards the domain of Hybrid Clouds.

REFERENCES

[1] [n. d.]. https://www.redhat.com/en/topics/devops/what-is-ci-cd

[2] [n. d.]. Continuous-Deployment. https://www.ibm.com/cloud/learn/continuous-deployment

[3] [n. d.]. What is multicloud? https://www.redhat.com/en/topics/cloud-computing/what-is-multicloud

[4] 2005. *An Architectural Blueprint for Autonomic Computing*. Technical Report. IBM.

[5] Y. Ahn, J. Choi, S. Jeong, and Y. Kim. 2014. Auto-scaling method in hybrid cloud for scientific applications. In *The 16th Asia-Pacific Network Operations and Management Symposium*. 1–4.

[6] Y. Ahn and Y. Kim. 2014. VM Auto-Scaling for Workflows in Hybrid Cloud Computing. In *2014 International Conference on Cloud and Autonomic Computing*. 237–240.

[7] C. Barna, H. Khazaei, M. Fokaefs, and M. Litoiu. 2017. Delivering Elastic Containerized Cloud Applications to Enable DevOps. In *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. 65–75.

[8] Jerome Boyer and Hafedh Mili. 2014. *Agile Business Rule Development: Process, Architecture, and JRules Examples*. Springer Berlin.

[9] Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. 2009. *Engineering Self-Adaptive Systems through Feedback Loops*. Springer Berlin Heidelberg, Berlin, Heidelberg, 48–70. https://doi.org/10.1007/978-3-642-02161-9_3

[10] Tao Chen, Rami Bahsoon, and Xin Yao. 2018. A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. *Comput. Surveys* 51 (06 2018). https://doi.org/10.1145/3190507

[11] Daniel Cukier. 2013. DevOps Patterns to Scale Web Applications Using Cloud Services. In *Proceedings of the 2013 Companion Publication for Conference on Systems, Programming, & Applications: Software for Humanity (SPLASH '13)*. Association for Computing Machinery, New York, NY, USA, 143–152. https://doi.org/10.1145/2508075.2508432

[12] A. Drumea and C. Popescu. 2004. Finite state machines and their applications in software for industrial control. In *27th International Spring Seminar on Electronics Technology: Meeting the Challenges of Electronics Technology Progress, 2004.*, Vol. 1. 25–29 vol.1.

[13] M. Guerriero, M. Ciavotta, G. P. Gibilisco, and D. Ardagna. 2015. A Model-Driven DevOps Framework for QoS-Aware Cloud Applications. In *2015 17th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. 345–351.

[14] Michael Httermann. 2012. *DevOps for Developers* (1st ed.). Apress, USA.

[15] Hyejeong Kang, Jung-in Koh, Yoonhee Kim, and Jaegyoon Hahm. 2013. A SLA driven VM auto-scaling method in hybrid cloud environment. In *2013 15th Asia-Pacific Network Operations and Management Symposium (APNOMS)*. 1–6.

[16] Jeffrey Kephart and D.M. Chess. 2003. The Vision Of Autonomic Computing. *Computer* 36 (02 2003), 41 – 50. https://doi.org/10.1109/MC.2003.1160055

[17] Y. Li and Y. Xia. 2016. Auto-scaling web applications in hybrid cloud based on docker. In *2016 5th International Conference on Computer Science and Network Technology (ICCSNT)*. 75–79.

[18] Daniela Loreti and Anna Ciampolini. 2015. Mapreduce over the Hybrid Cloud: A Novel Infrastructure Management Policy. In *Proceedings of the 8th International Conference on Utility and Cloud Computing (UCC '15)*. IEEE Press, 174–178.

[19] Marco Miglierina, Giovanni P. Gibilisco, Danilo Ardagna, and Elisabetta Di Nitto. 2013. Model Based Control for Multi-Cloud Applications. In *Proceedings of the 5th International Workshop on Modeling in Software Engineering (MiSE '13)*. IEEE Press, 37–43.

[20] D. Morn, L. M. Vaquero, and F. Galn. 2011. Elastically Ruling the Cloud: Specifying Application's Behavior in Federated Clouds. In *2011 IEEE 4th International Conference on Cloud Computing*. 89–96.

[21] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. 2018. Auto-Scaling Web Applications in Clouds: A Taxonomy and Survey. *ACM Comput. Surv.* 51, 4, Article 73 (July 2018), 33 pages. https://doi.org/10.1145/3148149

[22] Yar Rouf, Joydeep Mukherjee, Marios Fokaefs, Mark Shtern, Justin Le, and Marin Litoiu. 2019. Rule-Based Security Management System for Data-Intensive Applications. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON '19)*. IBM Corp., USA, 254–263.

[23] Y. Wadia, R. Gaonkar, and J. Namjoshi. 2013. Portable Autoscaler for Managing Multi-cloud Elasticity. In *2013 International Conference on Cloud Ubiquitous Computing Emerging Technologies*. 48–51.

[24] Indre Zliobaite, Albert Bifet, Mohamed Gaber, Bogdan Gabrys, Joao Gama, Leandro Minku, and Katarzyna Musial. 2012. Next Challenges for Adaptive Learning Systems. *SIGKDD Explor. Newsl.* 14, 1 (Dec. 2012), 48–55. https://doi.org/10.1145/2408736.2408746