# Adaptive Event Driven Infrastructure

João Campos
joao.g.campos@tecnico.ulisboa.pt

Instituto Superior Técnico, Lisboa, Portugal

October 2021

**Abstract**

Cloud computer systems have several challenges in the domains of resource sharing, allocation and administration.

With the advent of very large data centers and virtualization, operations such as creation and deletion of virtual resources have been subject of special attention concerning performance and efficiency. Automation of administrative tasks and dynamic configuration are primary goals in an effort to provide fast integration, testing and deployment of software and infrastructure.

The increase in complexity and scale of large systems has made clear that more help is needed for optimal configuration and management. This support can come from leveraging behavioral indicators from running systems with the application of optimization methods and machine learning algorithms.

This thesis introduces an architecture for integration of several systems for the collection and processing of data to facilitate the automated management of cloud infrastructures and describes the adopted strategy for implementation and test of the proposed model.

**Keywords:** Cloud Computing, Infrastructure, Adaptive Systems, Logs, Metrics, Events

## 1. Introduction

Since the beginning of computer science, resource sharing and allocation of hardware infrastructures have been a field of interest, research, and development. It had an impact in a wide range of computer science areas from the development of hardware and software all the way to Management practices and Governance policies in Information Technology (IT).

There are a variety of approaches and practices for architecture, management, and administration of shared infrastructure. Implementation choices are not solely based on capabilities. Sometimes cost, ease of integration, and versatility have a higher than expected weight in the selection of technologies. This requires management systems composed of large and diversified layers and modules.

The tasks of configuration and management of resources in large datacenters, is a complex one. Some of the primary goals are to minimize failures or maximize performance with the least possible amount of risk. This is especially difficult when there are a large number of systems composed of many modules.

The behavior of the system is key for the collection of relevant insight. Metrics, events, and logs are generated in large quantities. Exploration of this data may help in making administration decisions.

The goal of this thesis is to contribute to the dynamic management and automatic configuration efforts in a datacenter environment. It is proposed an architecture for automated infrastructure management. The suggested design includes a platform for applying optimizations techniques, a way of integrating multiple sources of data, and the possibility of making changes to existing infrastructure. There is also presented a working proof-of-concept of such platform, where a service is changed and modified in an automated manner.

## 2. Background

### 2.1. Virtualization

The application of the virtualization concept in computer systems has been extensive.

Virtual machines (VMs) make possible for several operating systems to run simultaneously on the same hardware. An hypervisor is a piece of software that manages the sharing of hardware resources and creates an isolated environment for each VM [1].

Network Virtualization helps in scaling, management and coexistence of several networks in the same infrastructure. Examples can go from simple Virtual Local Area Networks (VLANs) [2] to more complex Software-defined Networks (SDNs) [3] which facilitate management and simplify network creation and deletion.

Storage resource can also be separated from normal computing servers and made available as dedicated service over the network. They can then be shared among a large number of computers increasing availability and capacity. Software-defined Storage (SDS) aims to ease data configuration management by breaking the vertical alignment of conventional storage design [4]. It opens the possibility of security and priority rules in an end-to-end mechanism [5]. Nowadays there are several implementations of Software-Defined Storage and the concept has been a key component in large datacenters.

### 2.2. Cloud Model

Datacenters have evolved a lot since the times of the large computer rooms that housed the mainframe computers. In the last decades development has been made from the building construction and design to the practices and policies applied to manage such complex systems. Factors such as management constraints, time, and location, can lead to an unused surplus of capabilities. Cloud computing is model that promotes datacenter efficiency and allow easy resource sharing. Essential characteristics of cloud computing are *On-demand self-service*, *Broad network access*, *Resource pooling*, *Rapid elasticity*, *Measured Service* [6]. Virtualization of resources plays a key role in making those characteristics possible and functional.

### 2.3. Cloud Platforms

There are public and private clouds. Public clouds are offered commercially and are available for a large number of customers. Amazon Web Services, Microsoft Azure, and Google Cloud Platform are the major players [7] Private clouds are owned and operated by private entities, even though they can host several tenants. Large operators often have proprietary management platforms. However, there are some open source alternatives that help the management, control, and provision of the available infrastructure in accordance to the cloud model. Two of the most popular are OpenStack, CloudStack, and OpenNebula [8, 9].

### 2.4. DevOps

DevOps is a set of methodologies and practices that bring software development and IT operations closer together. It is facilitated by the softwarization of infrastructures.

Classical administration tasks, such as setting up virtual computing resources, traditionally involved a lot of manual work. With the increase of the number of machines to manage, automation of a large set of operations is a basic scalability requirement. Infrastructure as Code (IaC) is a method for specifying and provisioning infrastructure resources. Some principles of Iac are: *Systems can be easily reproduced*, *Systems are disposable*, *Systems are consistent*, *Processes are repeatable*, *Design is Always changing* [10]. Terraform is a tool that implements this concept [11].

Software tools ease the implementation of techniques and procedures in all components of DevOps.

Git [12] is a distributed version control software that tracks changes in files. It is useful for large teams working on the same code base. GitHub [13] and GitLab [14] are platforms that implement remote repositories along with features such as web hosting, project management, code testing and deployment. This makes them overall DevOps enabling platforms. Continuous integration and deployment, another concept and goal of DevOps, can be implemented using Gitlab CI/CD [15].

Configuration and management of VMs involves tasks such as installation of software or specific configurations. Chef is a tool for this. It uses a client/server configuration. This means an agent must be installed on the target system. Configuration is made in files called cookbooks written in the Ruby programming language.

### 2.5. Software Architectures

The advancements explored in the previous sections create excellent environments for complex applications to operate. There are some software architectural patterns that can provide some interesting improvements in development and deployment. Service-Oriented Architecture (SOA) is an architectural style where functionalities of a system are separated into services as opposed to being part of a monolithic application. Services usually expose an interface, and can communicate with a defined strict contract or defined protocol [16].

Microservices might be considered a modern take on SOA where services are more fine-grained. A microservice is a small program or functional element with a single responsibility. This fact improves maintainability and testing practices, which makes overall development easier. Microservices facilitate scalability and high availability, because more instances of the same functional element can be easily deployed. They can be developed more independently and in turn enable a higher frequency of releases and fixes while maintaining high reliability of the complex systems they form.

In the end these approaches might not be considered strict architectures but a style or paradigm that is used to implement a specific architecture.

### 2.6. Self-adaptive Systems

A self-adaptive system is a system that makes changes to itself and its behavior autonomously at run-time in response to changes in the environment. Implementation of these types of systems often involves using variations of the MAPE-K model pro-

posed by [17]. A representation of this model is presented in figure 1

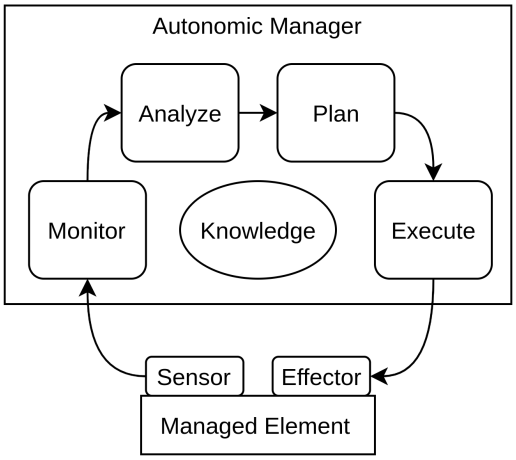


Figure 1: MAPE-K Model [17]

For an autonomic manager the capacity to monitor and execute present interesting challenges. Examples are collecting large volumes of data from different sources or integrating the executor with effectors. Administration of complexity IT systems is one example where this scenario exists.

Artificial Intelligence for IT Operations or AIOps offers an approach where artificial intelligence methods are used to help build and operate services more efficiently [18]. Implementations of this concept can be applied to anomaly detection, fault source identification or event correlation. Also they can offer optimizations, or improve performance by predicting resource utilization.

## 3. Related Work

There are some recent publications that show applications of the concepts of adaptive or intelligent systems, and datacenter automation. Each one implements, in a way, some goals of AIOps.

Sankie [19] is a project developed by Microsoft, in the Azure cloud environment. It is presented as a service to help developers increase velocity and throughput of changes and bug fixes. With a set of changes made by a developer the services can automatically recommend a reviewer for different areas of the source code. It can also identify a common subset of changes in files, functions, or modules and associate them to the most number of anomalies. The data was collected from sources in various stages of the development cycle. Some examples of sources are a git repository, alert logs, services logs, and performance indicators. Sankie uses the environment available in the Azure cloud for storage and computing.

In [20] the authors proposed an anomaly detection service. Their approach consisted in an pipeline with ingestion of data, model creation and alert systems. The techniques that yielded the best results were Spectral Residual outlier detector combined with convolutional neural networks (CNN).

In [21] the authors created a framework to detect impactful service system problems. Before applying data analysis they used a parser. It removed extra information like file IP addresses or file names. They created log sequences from logs with the same ID and removed duplicate events. After the log data exploration involved clustering techniques and correlation analysis.

To predict node failure in ultra-large-scale cloud computing platform (Amazon AWS) the authors of [22] used previous alert data of failures in combination with data about the location of nodes. The machine learning technique that got the best performance was random forests combined with oversampling of data from previous failed nodes.

In [23] is implemented an auto scaling platform based on deadlines and budget constraints. Their architecture consists on a decider that collects historical data from a repository, and upon analysis sends a plan to a module they call VM manager. This manager interacts with the Azure Cloud services to scale out servers.

An autoscaling service that enables scaling of applications in a cloud environment is presented in [24]. Applications can be scaled based on defined metrics such as cpu or memory consumption. The architecture is based on microservices. This way scaling monitoring and controller are separated and communicate with API calls. A test implementation is shown working for the cloud platform IBM Bluemix, now called IBM Cloud.

## 4. Solution Architecture

The proposed solution consists of a platform where data is collected, analyzed, and leveraged to implement adjustments on certain systems. It is an application of the MAPE-K control loop, where the analysis, planning, and execution functions exist inside a platform. The name chosen for this new platform is *OpsManager*. It will help in configuring and manage systems that from now on will be refereed as *target service* or *target system*.

Figure 2 shows how the platform fits into a datacenter environment. The *Infrastructure Platforms* element represents the collection of systems that support infrastructure creation, management, as well as development. Some examples are cloud platforms or repositories where *infrastructure as code* is stored and deployed. They support the lifecycle of a target system. *Data platforms* are services that store event and log data generated by the target system. OpsManager integrates with these existing

systems by consuming data and generating configuration changes, thus closing the cycle of adaptive infrastructure.

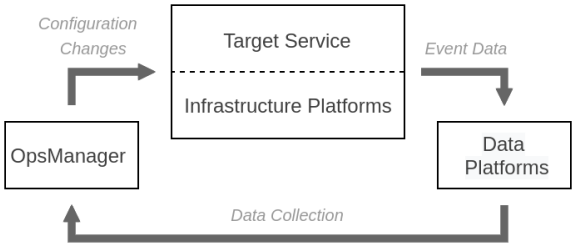


Figure 2: Proposed Solution Architecture

The OpsManager internal architecture is highly modular. This approach meant decoupling functions and services. Components can be developed to fit the implementation environment. They can also be cloned to provide resilience or for scaling purposes. This design choice is based on concepts taken from SOA and microservices.

In figure 3 the architecture of the platform can be observed.

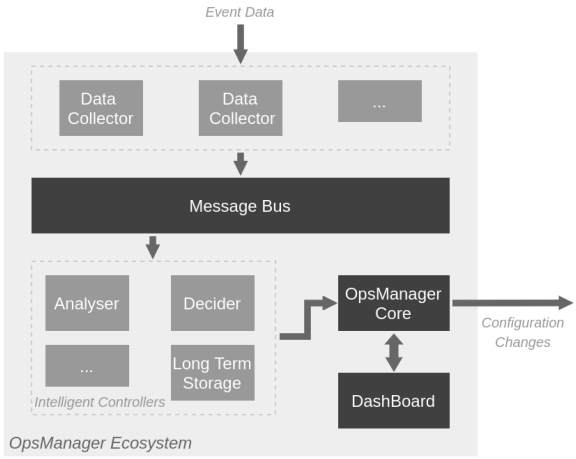


Figure 3: OpsManager Architecture

The components include the OpsManager Core for execution of changes, a message bus for module communication, data collectors, the intelligent control group modules for analysis and decision making, and a dashboard that provides a user interface.

Collector functions can be programmed independently for each source of data. Collected data may be needed by other concurrent microservices at the same time. If the data increases in volume and requires parallel processing the message bus facilitates partitioning and horizontal scaling. The possibility of adding new target services, with different data structures and sources reinforces these design choices. The other dotted box below with the caption *Intelligent Controllers* encompasses more microservices. An analyzer may implement an algorithm to extract insight from data and a decider might suggest or trigger an infrastructure change. Long term storage may store historical data that will help in forecasting. The dashboard might also receive streaming data and information from the microservices and for example display it in a graph. The microservices should also be able to interact with OpsManager Core via an API.

OpsManager core was created out of the necessity of having a way of selecting which configuration changes to apply and providing a backend for a user interface. It has plugins to interact with infrastructure platforms. A decider microservice invokes an endpoint sending infrastructure modifications suggested by its algorithm. OpsManager core then is programmed to wait for user confirmation or apply this automatically. This helps in keeping those microservices fairly simple.

Data visualisation is helpful in conveying the status of a certain system. This is one of the main goals of having a dashboard. The other is enabling a human administrator to have control over certain aspects of infrastructure management. The administrator might be presented with several suggestions of modifications and only apply one, or set a certain decider as automatic and authoritative over a target system.

## 5. Implementation

A proof-of-concept implementation was made in the datacenters of Direcção dos Serviços de Informática (DSI) at Instituto Superior Técnico. For the purpose of this thesis it was provided a specific environment. The technologies present were required to be used and integrated in this proof-of-concept.

### 5.1. Environment

Openstack [25] is the provider where a target system must be instantiated. Terraform [26] is used to define the infrastructure as code. Besides describing virtual machines, networks, and storage, it can also be utilized for other things such as key generation for each deployment. A centralized secret or token storage platform is available and implemented with the software Vault [27].

Once a VM is created it must be provisioned with Chef. There is a Chef server where configuration guides called recipes can be stored and fetched. All files from Chef and Terraform are version controlled in GitLab. A Group in GitLab may include repositories for the infrastructure code, the provisioning code, the software of a target service, and the deployment definitions with pipelines that create or destroy infrastructure in test and production environments. The is how Gitlab enables *Continuous Integration* and *Continuous Delivery*.

Once a target system is deployed it may be necessary to advertise its existence in a central registry. This service is supported by Consul [28].

The target service might expose a port for incoming external connections. For example a web service listening on an HTTP or HTTPs port. An Nginx [29] server exists in this environment and acts as a reverse proxy. This reverse proxy provides statistics about each backend server such as total requests and response times. There is also an instance of Graylog [30] for the collection of text logs from servers.

### 5.2. Target Service

The target service to be modified consists of a web application with a variable number of backend servers.

For the purpose of this thesis DSI provided a base template of a target service that included Terraform infrastructure and Chef recipes to instantiate and provision nodes in two datacenters. A GitLab group also existed with deployment pipelines. Each pipeline had "spin-up" and "destroy". Infrastructure was modified so that consecutive pipelines could be triggered where "spin-up" only made changes and destroy was not needed to be executed before the next pipeline.

Terraform is configured to store its state in a backend location accessible to all jobs. Any execution will take the last state and work the changes from there.

Part of the provisioning of a server included the generation of a policy file for each unique server configuration. The policy file includes all recipes and their versions to be applied. This file is uploaded to the common chef server. The pipeline was modified to include verification of existing policies for the same deployment and to not override previous ones.

SSH keys for the servers are generated by Terraform inside the job "Spin Up". If a new pipeline instantiates a new server, it must have the same keys as the old ones. Since a central secret storage using Vault was available, it was used to keep these keys accessible through infrastructure iterations.

Horizontal scaling happens when more nodes are added or subtracted. This was the modification goal. Environment variables were set to indicate the number of backend nodes desired. This makes it possible to scale an infrastructure without having to hardcode changes to Terraform files. There was only the need to create triggers in GitLab to initiate the pipelines.

A web service to be deployed to the nodes was programmed so that each request generates a specific predetermined computational load. Concurrent requests will slow down the average response time.

### 5.3. OpsManager

Figure 4 shows the architecture implemented along with the environment integration.

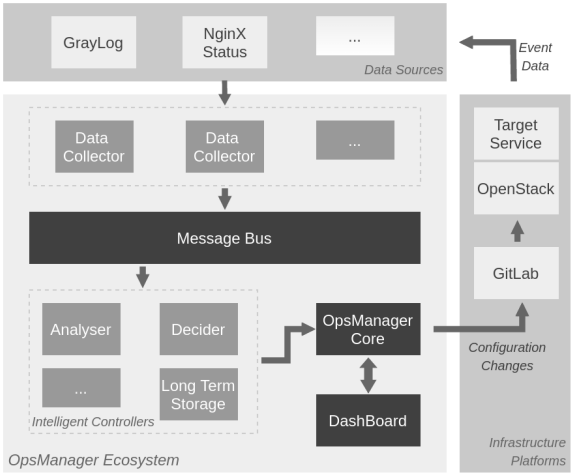


Figure 4: OpsManager Implementation

On top several data sources are given as an example of different types of information that can be collected. These can be simple metrics (NingX Status) or text logs and events (Graylog).

On the right side infrastructure platforms that OpsManager may interact with directly. The example given shows the situation previously described where GitLab contains code, infrastructure definitions and provision files as well as CI/CD pipelines that create deployments in OpenStack. The arrows demonstrate a flow where information about a service is collected, analyzed and how it generates changes applied to the infrastructure.

NingX Status was used as a data source to create the adaptive cycle. It was created a collector that filters and transforms the data as needed.

The NingX Status module contains virtual host status information. The data available is extensive. In an environment with about 50 servers divided into zones, the data can easily reach 200 kB with an average of 4 kB per server.

The collector pulls a json format response. The data is broken down into messages and sent to the message bus. For this specific target service the information needed is the server response times, request counter and error codes. After filtering the needed data the collector adds a current timestamp and sends the final message the queue in the message bus.

Apache Kafka was used for the message bus [31]. It provides messaging queuing, storage, redundancy and partitioning of data. It is mostly used for real-time streaming data pipelines for its high through-

put and scalability.

A decider microservice consumes already prepared data and uses it to make decisions about infrastructure. A simple moving average algorithm was chosen. This is an average of the response times of a defined time window, for example the last 10 minutes calculated every minute. It is applied to response times of all servers in an *upstreamZone*. Since the service is a simple web service with a predetermined fixed computational cost, multiple concurrent requests on the same server will increase the average response time.

Infrastructure changes are separated with a defined interval, so that the system has time to stabilize. If the moving average of response times is above a certain threshold a scale up operation is suggested. The simplified evaluation expression is $(average\_response\_time \geq up\_threshold)$. When more node are added to the service the response time is expected to decrease, if not more nodes will be added after the aforementioned defined interval.

When a scale up happens, the request counter average is stored. This value is important as the scale down operation cannot be triggered by the decrease in average of response time, after all it is the goal of the scale up. So the algorithm compares the current number of requests with the number of requests that indirectly triggered the last scale up. If the current level is below this number multiplied by an adjusted $\alpha$ parameter it means that the excessive load is no longer present so a new scale down operation is suggested. The simplifeid evaluation expressin is $(current\_average\_requests \leq last\_up\_average\_requests.peek() \times \alpha)$, where last_up_average_requests is the stack like data structure that stores request counter average at scale up times. In the end scale ups have a fixed threshold but scale downs are defined at runtime.

The internal structure of OpsManager Core allows and expects the use of plugins to interact with infrastructure or code platforms such as GitLab. This means that if there is a migration from GitLab to another platform say GitHub, there is only the need to create another plugin. It was created the necessary code to trigger GitLab pipelines upon a simple endpoint invocation.

## 6. Results

The scenario for tests is composed of the target service described in 5.2, that running with one backend server. Several request are made to this service with the objective of increasing the average response time. The idea was to simulate a usage in working hours with levels of low, medium, and high intensity throughout the day.

Figure 5 shows the response of the system. The time window chosen is 10 minutes.

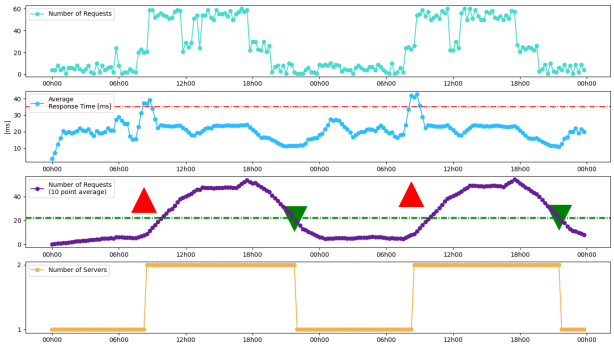


Figure 5: Target Service Scaling

The top plot axis shows total requests made in each interval. The second the average response time, which is used to decide scale ups. This threshold is identified by the red horizontal line at 35 ms. The third axis shows the 10 point average of the number of requests. This average decides the scale downs. Since scale downs are computed from the request numbers at scale up they have different thresholds identified by the green horizontal lines. The red upward and green downward triangles show the scale up and scale down moments respectively. The number of active servers at each time can be confirmed in the last axis. The reaction and result of the system can be observed when the load increases. Average response time surpasses the red threshold and another server node is added. In the few next points the average response decreases accordingly.

With these results we can infer that that internal and external integration of OpsManager works as expected.

## 7. Achievements

The possibilities offered by the cloud computing model have and will continue to shape the way information systems are implemented and operated.

Developments in artificial intelligence and machine learning will have more and more influence on the way large and complex systems are managed. It is therefore important to have the means to easily implement and test new algorithms and techniques.

One of the goals of this thesis was to provide the first steps towards a management platform architecture to develop an event driven adaptive infrastructure. The developed work accomplished this by designing and proposing a solution that easily interconnects multiple systems and platforms. The results show capabilities for data collection, decision making and infrastructure modification.

## 8. Future Work

The OpsManager platform developed, provides the basis of an adaptive, event driven infrastructure.

Although the proof-of-concept presented here

provides a model for the overall system, there are several improvements that are advisable to reach an operational state.

The continuation of the development of the dashboard is certainly an important goal. With it comes multi-user capabilities and secure access. Microservices platforms could be used to facilitate their deployment.

For target services a preloaded node image with all the required software might be one approach. Caching of software used in the nodes is another possibility. The use of unikernels or other lightweight virtualization methods instead of whole virtual machines, may provide improvements.

In the end the use of the platform should now make possible to start exploring more complex optimizations algorithms and machine learning models.

**References**


[1] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. Hypervisor: A survey on concepts and taxonomy. *International Journal of Innovative Technology and Exploring Engineering*, 2(3):222–225, 2013.

[2] Jorge Carapinha and Javier Jiménez. Network virtualization: a view from the bottom. In *Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*, pages 73–80, 2009.

[3] Paul Goransson, Chuck Black, and Timothy Culver. *Software defined networks: a comprehensive approach*. Morgan Kaufmann, 2016.

[4] Ricardo Macedo, João Paulo, José Pereira, and Alysson Bessani. A survey and classification of software-defined storage systems. *ACM Computing Surveys (CSUR)*, 53(3):1–38, 2020.

[5] Eno Thereska, Hitesh Ballani, Greg O'Shea, Thomas Karagiannis, Antony Rowstron, Tom Talpey, Richard Black, and Timothy Zhu. Ioflow: A software-defined storage architecture. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 182–196, 2013.

[6] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[7] Rachel Liu Deepak Mohan, Andrew Smith. Idc marketscape: Worldwide public cloud infrastructure as a service 2020 vendor assessment, September 2020.

[8] Salman A Baset. Open source cloud technologies. In *Proceedings of the Third ACM Symposium on Cloud Computing*, pages 1–2, 2012.

[9] P Bedi, B Deep, P Kumar, and P Sarna. Comparative study of opennebula, cloudstack, eucalyptus and openstack. *International Journal of Distributed and Cloud Computing*, 06(01):37–42, June 2018.

[10] Kief Morris. *Infrastructure as code: managing servers in the cloud.* " O'Reilly Media, Inc.", 2016.

[11] Leonardo Rebouças de Carvalho and Aleteia Patricia Favacho de Araujo. Performance comparison of terraform and cloudify as multicloud orchestrators. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 380–389. IEEE, 2020.

[12] Git. Git about. Git. Acessed 3 Oct 2021.

[13] GitHub. Github about. GitHub. Acessed 3 Oct 2021.

[14] GitLab. Gitlab about. GitGitLab. Acessed 3 Oct 2021.

[15] Charanjot Singh, Nikita Seth Gaba, Manjot Kaur, and Bhavleen Kaur. Comparison of different ci/cd tools integrated with cloud platform. In *2019 9th International Conference on Cloud Computing, Data Science & Engineering (Confluence)*, pages 7–12. IEEE, 2019.

[16] David Sprott and Lawrence Wilkes. Understanding service-oriented architecture. *The Architecture Journal*, 1(1):10–17, 2004.

[17] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[18] Anton Gulenko, Alexander Acker, Odej Kao, and Feng Liu. Ai-governance and levels of automation for aiops-supported system administration. In *2020 29th International Conference on Computer Communications and Networks (ICCCN)*, pages 1–6. IEEE, 2020.

[19] Rahul Kumar, Chetan Bansal, Chandra Maddila, Nitin Sharma, Shawn Martelock, and Ravi Bhargava. Building sankie: An ai platform for devops. In *2019 IEEE/ACM 1st International Workshop on Bots in Software Engineering (BotSE)*, pages 48–53. IEEE, 2019.

[20] Hansheng Ren, Bixiong Xu, Yujing Wang, Chao Yi, Congrui Huang, Xiaoyu Kou, Tony Xing, Mao Yang, Jie Tong, and Qi Zhang. Time-series anomaly detection service at microsoft. In *Proceedings of the 25th ACM*

*SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 3009–3017, 2019.

[21] Shilin He, Qingwei Lin, Jian-Guang Lou, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 60–70, 2018.

[22] Yangguang Li, Zhen Ming Jiang, Heng Li, Ahmed E Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. Predicting node failures in an ultra-large-scale cloud computing platform: an aiops solution. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 29(2):1–24, 2020.

[23] Ming Mao, Jie Li, and Marty Humphrey. Cloud auto-scaling with deadline and budget constraints. In *2010 11th IEEE/ACM International Conference on Grid Computing*, pages 41–48. IEEE, 2010.

[24] Seetharami R Seelam, Paolo Dettori, Peter Westerink, and Ben Bo Yang. Polyglot application auto scaling service for platform as a service cloud. In *2015 IEEE International Conference on Cloud Engineering*, pages 84–91. IEEE, 2015.

[25] OpenStack Foundation. Introduction: A bit of openstack history. Microsoft Corporation - Documentation. Accessed 27 Dez 2020.

[26] HashiCorp. Introduction to terraform. HashiCorp. Accessed 1 Oct 2021.

[27] HashiCorp. Introduction to vault. HashiCorp. Accessed 1 Oct 2021.

[28] HashiCorp. Introduction to consul. HashiCorp. Accessed 1 Oct 2021.

[29] NGINX. Nginx reverse proxy. NGINX Docs. Accessed 1 Oct 2021.

[30] GRAYLOG. About graylog. GRAYLOG. Accessed 1 Oct 2021.

[31] Apache Kafka. Kafka about. Apache Kafka. Acessed 3 Oct 2021.