VeriFogOps: Automated Deployment Tool Selection and CI/CD Pipeline Generation for Verifying Fog Systems at Deployment Time

Hiba Awad, Thomas Ledoux, Hugo Bruneliere, Jonathan Rivalan

VeriFogOps: Automated Deployment Tool Selection and CI/CD Pipeline Generation for Verifying Fog Systems at Deployment Time

Hiba Awad
IMT Atlantique, LS2N (CNRS), Inria Rennes, Smile
Asnières-sur-Seine and Nantes, France
hiba.awad@smile.fr

Thomas Ledoux
IMT Atlantique, LS2N (CNRS), Inria Rennes
Nantes, France
thomas.ledoux@imt-atlantique.fr

Hugo Bruneliere
IMT Atlantique, LS2N (CNRS)
Nantes, France
hugo.bruneliere@imt-atlantique.fr

Jonathan Rivalan
Smile
Asnières-sur-Seine, France
jonathan.rivalan@smile.fr

Abstract

Fog Computing consists in decentralizing the Cloud by geographically distributing computation, storage, network resources, and related services. Among other benefits, it allows reducing bandwidth usage, limiting latency, or minimizing data transfers. However, Fog systems engineering remains challenging and quite often error-prone. Following best practices in software engineering, verification tasks can be performed before such systems are concretely deployed. Works already exist on verifying non-functional properties of Fog systems at different previous steps of the life cycle. This paper goes one step further and presents the VeriFogOps approach. This approach allows to automatically select deployment tools, based on expressed Quality of Service (QoS) requirements, and then generate relevant CI/CD pipelines supporting the deployment of Fog systems. We implemented and validated our approach via two realistic use cases, considering different QoS solutions and deployment tools. This work, developed in direct collaboration with our industrial partner Smile, goes towards the direction of a more comprehensive support for the entire life cycle of Fog systems, from design to actual deployment and execution.

CCS Concepts

• **Software and its engineering → Software verification and validation**; **Software development process management**; • **Computer systems organization → Cloud computing**.

Keywords

Verification, Deployment, CI/CD pipeline, Automation, Quality of Service, DevOps, Fog Computing

ACM Reference Format:
Hiba Awad, Thomas Ledoux, Hugo Bruneliere, and Jonathan Rivalan. 2025. VeriFogOps: Automated Deployment Tool Selection and CI/CD Pipeline Generation for Verifying Fog Systems at Deployment Time. In *The 40th ACM/SIGAPP Symposium on Applied Computing (SAC '25), March 31-April 4, 2025, Catania, Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3672608.3707854

SAC '25, March 31-April 4, 2025, Catania, Italy

1 Introduction

Fog computing [16, 31] is a paradigm aiming to decentralize the Cloud by geographically distributing computation, storage and network resources, as well as related services. It is at the crossroads of complementary areas such as Cloud computing [4], Edge computing [23], and IoT [15]. In practice, large data centers in core networks, and micro data centers or computing-enabled devices at the edge of networks, are used in a collaborative way within a single large-scale geo-distributed system called *Fog system*.

Recent work has shown that Fog systems are often very difficult and expensive to engineer and then manage [1, 16]. While some works already target the verification of properties on Fog systems at different steps of the life cycle [5, 10, 30], it is still challenging to ensure an appropriate Quality of Service (QoS) [29] when configuring these Fog systems before their deployment. This is notably due to their distributed and heterogeneous nature, limited computational resources, and dynamic environments. These characteristics make it difficult to efficiently handle resource allocation, security aspects, latency control or reliability (among others). Thus, an important objective is also to allow the verification of QoS requirements at deployment time. This was confirmed by the feedback we received from our industrial partner Smile, based on their own experience in customer projects. To this end, following best practices in software engineering, a key capability is to automate the selection of the most suitable deployment tool and related configuration according to provided QoS requirements. Moreover, Fog systems can target many different application domains and can possibly rely on various deployment solutions. As an answer, in this paper we propose the VeriFogOps generic approach for deployment tool selection and automated CI/CD pipeline generation.

To fulfill this objective, the proposed iterative approach mostly contributes two new components that rely on 1) The use of CI/CD pipeline, combined with different types of (*deployment tools* and *QoS solutions*) couples, to select the most appropriate deployment tool and corresponding deployment configuration file for a given Fog system, 2) The use of this deployment configuration file, along with associated deployment instructions, in order to automatically

generate a CI/CD pipeline that enables the verification of QoS requirements and the automated deployment of this Fog system via the selected deployment tool. To evaluate our approach, we provide an implementation that we validate in the context of three different use cases from the literature. On these use cases, we consider two different types of QoS solutions (related to security issues and deployment costs respectively) with different deployment configuration files. As a result, and based on various deployment instructions, we show that we are able to automatically select the most suitable deployment tool and corresponding configuration as well as to generate the related CI/CD pipeline.

The paper is organized as follows. Section 2 presents the general background and motivation for our work, resulting in three research questions. Section 3 introduces the overall VeriFogOps approach. Then, Section 4 focuses on the automation support we propose for deployment tool selection and CI/CD pipeline generation. Section 5 illustrates the practical validation we performed in the context of three different use cases from the literature. Then, Section 6 summarizes the current Gitlab-based implementation of the proposed approach and implementation. Section 7 discusses the current capabilities and limitations of our work. Finally, Section 8 elaborates on the related work, while Section 9 concludes the paper.

2 Background and Motivation

As introduced in Section 1, verifying QoS requirements after deploying Fog systems can quickly become challenging and expensive. As a consequence, it is necessary to consider alternative approaches to better support the verification of QoS requirements before the actual deployment of these systems.

This idea is directly reflected by the current Shift-Left trend in the software industry [25]. The Shift-Left approach advocates for the identification and resolution of bugs as early as possible during the engineering process. A practical illustration of the Shift-Left approach is the DevSecOps paradigm [18] which intends to deal with the security concerns before operation processes. The overall objective is to improve the quality of the engineered systems while reducing the time spent to resolve problems later in the system's life cycle. Going in this direction, recent work provided some approaches for verifying QoS requirements during different engineering phases by relying on various techniques [5, 13, 33]. However, up to our current knowledge, there are only few existing approaches verifying QoS requirements at deployment time.

In the industry, several solutions have already been developed to tackle deployment-related QoS requirements. Indeed, the ecosystem is rich and quite heterogeneous in terms of deployment tools (e.g., *Terraform*[1], Kubernetes[2] *Ansible*[3], *Chef*[4]) and associated QoS solutions. For instance, the *InfraCost*[5] motto is "Shift FinOps Left With Infracost": it enables a Shift-Left (i.e., prior to deployment) approach to infrastructure costs by providing cost estimates for *Terraform*, an infrastructure-as-code tool. Another example is *tfsec*[6] which provides a security scanner based on the static analysis of Terraform code. Because of this diversity, there is a risk of being limited by or even locked in a particular deployment tool.

As a consequence, and as confirmed by the experience of our industrial partner Smile, there is the need for a global generic approach that support such a rich deployment ecosystem in an automated way. The popular DevOps paradigm already advocates for the automation of both development and operation activities [12]. To support this, CI/CD pipelines are essential automation components [2]. Such pipelines consist in series of steps automating the software workflow by building, testing and deploying the corresponding code in a regular and iterative way. In our context, obtaining CI/CD pipelines well-adapted to our QoS requirements verification needs appear to be fundamental [32].

Based on this overall analysis, we propose in this paper to address the following Research Questions (RQs):

- **RQ1** How to support in a generic way a rich and heterogeneous DevOps ecosystem while verifying QoS requirements before deployment?
- **RQ2** Which automation components (e.g., roles, pipelines, languages) are needed to support this DevOps ecosystem and QoS requirement verification?
- **RQ3** How to validate these automation components and corresponding generic approach for different QoS requirements across various Fog application domains?

3 An Approach for Deployment Tool Selection and Automated CI/CD Pipeline Generation (RQ1)

As introduced in Section 1, the objective of the VeriFogOps approach is twofold. First, it aims at supporting the automated selection of the most relevant deployment tool and deployment configuration file, according to QoS requirements expressed by a Fog System Architect (FSA) in charge of the concerned Fog system. Second, it aims as using the result of this selection in order to automatically generate a corresponding CI/CD pipeline, that verifies these QoS requirements before actually performing the Fog system's infrastructure deployment. During the process, the FSA collaborates with both DevOps engineers (i.e., development experts) and SysOps engineers (i.e., system experts). Figure 1 provides an overview of the VeriFogOps approach.

⓪ The *Fog System Architect* (FSA) is the main actor. Ideally, this role should be held by an engineer or a group of engineers having knowledge and experience in terms of both Fog infrastructures (including their non-functional characteristics) and the targeted application domains (smart cities, industrial IoT, etc.). Thus, the FSA is notably in charge of specifying the QoS requirements for the concerned Fog system.

① The FSA begins using the *Deployment Tool Selection* component by specifying *Deployment instructions* within one single declarative file. In this file, the FSA specifies the different QoS requirements expected to be verified before the Fog system's deployment. Associated rules, selection priorities and corresponding deployment status are also indicated at this step.

② Then, the FSA specifies (from scratch) or retrieves (from other solutions such as VeriFog [5, 6]) various *Deployment configuration* files wished to be verified. The FSA also identifies the various

[1] https://www.terraform.io/
[2] https://kubernetes.io/
[3] https://www.ansible.com/
[4] https://www.chef.io/
[5] https://www.infracost.io/
[6] https://aquasecurity.github.io/tfsec/

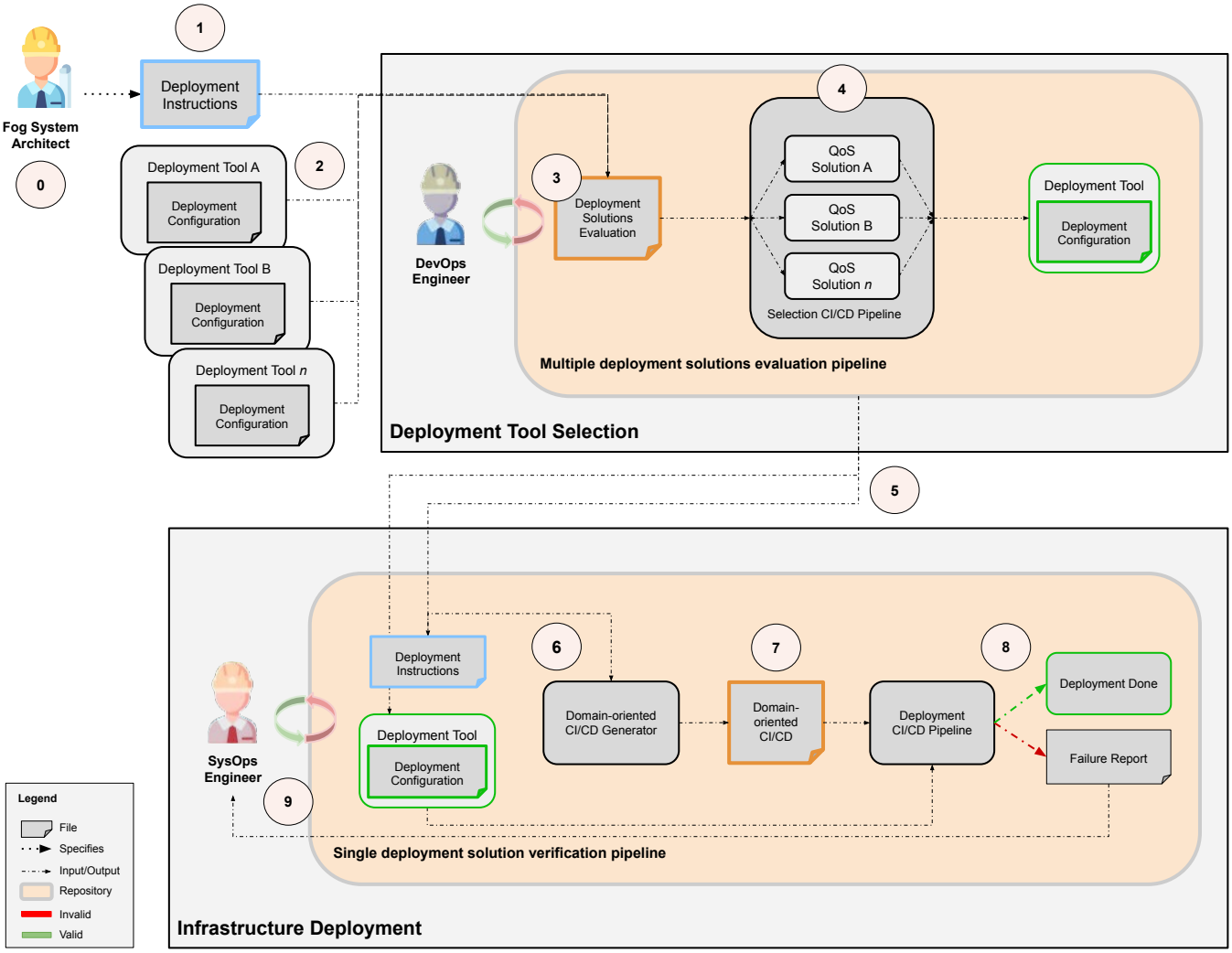

Figure 1: Overview of the VeriFogOps approach, deployment tool selection and automated CI/CD pipeline generation for Fog systems (the circled numbers correspond to numbered paragraphs in the text).

Deployment tools to be considered within the selection process. These tools are used to actually deploy the system infrastructure according to specific types of *Deployment configuration* files.

③ The *DevOps Engineer* then comes into play, as an expert in deployment tools and corresponding QoS solutions for verifying QoS requirements. The DevOps engineer examines the *Deployment configuration* files and associated *Deployment tools* (proposed by the FSA) to identify appropriate *QoS solutions*. These solutions are designed to ensure that a given deployment configuration meets the expressed QoS requirements. Each type of *Deployment configuration* file / *Deployment tool* comes with associated *QoS solutions* dedicated to particular kinds of requirements. As a result, combinations are formed (e.g., *Deployment tool A*, *QoS solution A*) and described in a repository within a CI file called *Deployment Solutions Evaluation*. This choice of *QoS solutions* is directly guided by the QoS requirements specified in the *Deployment instructions*. It can also be influenced by technical constraints, such as the availability of a given *QoS Solution* for a particular *Deployment tool*. Note that the engineering effort from the DevOps engineer may be required only once, e.g., when the architecture of the Fog system is first defined. This effort can also be reused over multiple projects, provided that they share common QoS requirements and deployment tools.

④ The *Deployment solutions evaluation* created by the DevOps engineer is executed by a *CI/CD Pipeline* having three stages. The "build" stage sets the appropriate environment to run the different verification tests. It also ensures that the deployment configuration files exist and are not empty. The "test" stage verifies the selection-related QoS requirements specified by the FSA in the *Deployment instructions* file. The output of this stage is then analysed in the last "select" stage. At this point, the *CI/CD Pipeline* automatically identifies the *Deployment tool* and corresponding *Deployment configuration* file providing the best results.

⑤ After selecting the most appropriate *Deployment Tool* and *Deployment configuration* file, the resulting information is transferred to the *Infrastructure Deployment* component along with the *Deployment Instructions* originally specified by the FSA.

⑥ In this component, the *Domain-oriented CI/CD Generator* takes the *Deployment Instructions* as input in order to generate a corresponding *Domain-oriented CI/CD* descriptor file. Such a *Domain-oriented CI/CD Generator* can be notably based on templates targeting several different deployment tools and QoS requirements. It can also be further extended according to the needs.

⑦ The generated *Domain-oriented CI/CD* file describes the three stages of the *CI/CD Pipeline* to be executed. The "build" stage presents the environment to run the verification of the QoS requirements indicated by the FSA. Then, the "test" stage describes the verification tests to be performed, along with their rules and deployment status as specified in the *Deployment Instructions*. Finally, the "deploy" stage indicates the actual *Deployment Configuration* file to be deployed with the selected *Deployment Tool*.

⑧ From the *Domain-oriented CI/CD* file, a *CI/CD Pipeline* is automatically created for a specific CI/CD platform. This *CI/CD Pipeline* then automatically executes all the stages specified in the *Domain-oriented CI/CD* file. After execution, there are two possibilities depending on the obtained results and tests status. In the first case, we have a *Deployment Done*: all mandatory tests successfully passed, or there were no mandatory tests. Similarly, all other stages have been successfully completed, so the pipeline is valid and the deployment is considered a success. In the second case, the pipeline stopped and a *Failure Report* is produced: at least one of the mandatory tests failed. For example, a given value exceeded one of the thresholds mentioned in the *Deployment Instructions* file.

⑨ Finally, a *SysOps Engineer* can check this *Failure report* and react accordingly. He is an expert in deployment tools and infrastructure constraints who is able to manually edit the *Deployment Configuration* files whenever needed. Depending on the type of failure, the SysOps engineer can decide 1) to directly modify the *Deployment Instructions* file, by changing the rules or status of the failed test, in order to maintain the pipeline and obtain a successful deployment, or 2) to change the selected *Deployment configuration* file, eventually by also triggering another iteration of the *Deployment Tool Selection* process. Although a collaboration may occur between the SysOps engineer and the FSA in case of a failing deployment, the former is able to autonomously modify the configurations. The performed updates can allow the reconciliation between the intent of the original *Deployment Configuration* and newly appeared requirements concerning the targeted deployment environment (e.g., prices, security policies or API changes). Such an evolution may be quite frequent and sometimes unforeseen.

4 Language and Automation Components Dedicated to Deployment Tools (RQ2)

We now focus on the main components we propose in the VeriFogOps approach to support interactions with configuration files at deployment time. In what follows, we provide more details on the simple language we designed for expressing the required *Deployment Instructions*. Then, we describe further the two core capabilities of VeriFogOps, namely the automated selection of deployment tools and the generation of corresponding CI/CD pipelines.

4.1 Deployment Instructions

In VeriFogOps, we propose a simple declarative language for architects and engineers to describe the QoS requirements to be 1) considered for deployment tool selection and 2) verified before actual deployment. All this information is encapsulated within a given file called *Deployment Instructions*. Having such a single file notably allows for an easier sharing during the whole deployment preparation process. As presented in Section 3, the main purpose of the *Deployment Instructions* file is for the FSA to specify QoS requirements, associated rules, selection priorities and deployment status. With these information, our components can then verify and guarantee before actual deployment that the selected *Deployment Configuration* file can satisfy the expressed QoS requirements.

To introduce the proposed language, Figure 2 shows an example of a basic *Deployment Instructions* file describing verification instructions for three main QoS requirements relevant for Fog systems: configuration syntax, security and cost.

```yaml
tests:
  - name: ConfigurationSyntax
    rules:
      MAX_LINT_ISSUES: 155
    deployment_status: Blocking

  - name: Security
    rules:
      CRITICAL_PROBLEMS: 17
      HIGH_PROBLEMS: 3
      MEDIUM_PROBLEMS: 4
      LOW_PROBLEMS: 5
    selection_priority: 1
    deployment_status: Optional


  - name: Cost
    rules:
      BUDGET: 100
    selection_priority: 2
    deployment_status: Optional
```

Figure 2: Excerpt of the YAML textual notation of the proposed Deployment Instructions language.

Each individual instruction has a *name* and can be expressed by a set of *rules* specifying the value of different CI/CD variables. For example, such a variable can be a threshold for the maximum number of security problems allowed (e.g., *CRITICAL_PROBLEMS*). The *selection_priority* indicates whether the concerned QoS requirement has to be considered (or not) for the deployment tool selection process. When provided, the corresponding value indicates the preference in case of more than one suitable deployment tool (i.e., 1 is more priority than 2). For example, if a deployment tool A excels in cost efficiency while deployment tool B offers a superior security level, the *selection_priority*, where a lower value indicates a higher priority, will determine the final choice. Finally, the *deployment_status* element indicates whether the QoS requirement check is "optional" or "blocking" for deployment. In the first case, an invalid check does not prevent the rest of the deployment pipeline

from being executed. In the second case, it directly stops execution of the deployment pipeline.

4.2 Deployment Tool Selection

As presented in Section 3, the main purpose of the *Deployment Tool Selection* component is to verify QoS requirements for different *Deployment Configuration* files corresponding to various types of *Deployment tools* and using different *QoS solutions*. To this end, a DevOps engineer takes as inputs the *Deployment configuration* files and the *Deployment instructions* in order to build a CI file called *Deployment Solutions Evaluation*. This file mostly specifies the set of stages for the CI/CD pipeline to build, test and select the most suitable *Deployment Tool*. Based on these inputs and the expertise of the DevOps engineer, different *QoS Solutions* are then investigated corresponding to the expressed QoS requirements. For example, if the *Deployment Tool* is DT1, and the verification test specified in the *Deployment Instructions* is cost, the DevOps engineer can specify in the *Deployment Solutions Evaluation* file DT1:cost as a *QoS Solution* for corresponding *Deployment configuration* files. This way, the DevOps engineer can indicate in the *Deployment Solutions Evaluation* file different *Deployment Tool – QoS solution* pairs for verifying specific QoS requirements. After that, this file is ready to provide and execute a *Selection CI/CD Pipeline*. Figure 3 shows this pipeline and its three main stages ("build", "test" and "select") considering Cost and Security as QoS requirements.

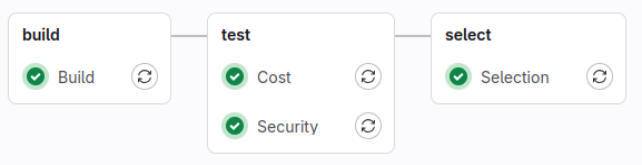

Figure 3: Deployment Tool Selection Pipeline.

This pipeline first verifies in the "build" stage the existence of the *Deployment Configuration* files. In the "test" stage, each of the indicated *QoS Solutions* is automatically executed on the corresponding *Deployment Configuration* file. The results produced by each *QoS Solution* are then analyzed and compared against each other. Finally, in the "select" stage, the pipeline determines the most suitable *Deployment Tool* and corresponding *Deployment Configuration* file according to the previous results and selection priority.

4.3 CI/CD Generation

Once the most appropriate *Deployment Tool* selected, the corresponding *Deployment Configuration* and *Deployment Instructions* files trigger the *Domain-oriented CI/CD Generator*. Based on these files, the *Domain-oriented CI/CD Generator* produces a *Domain-oriented CI/CD* file by combining several embedded templates. As a result, it creates a corresponding *Deployment CI/CD Pipeline* that automatically executes the required stages.

Figure 4 shows the resulting pipeline composed of three main stages: "build", "test", and "deploy". In the "build" stage, the *Deployment configuration* file – selected by the *Deployment Tool Selection* component – is first injected. Then, the "test" stage performs the

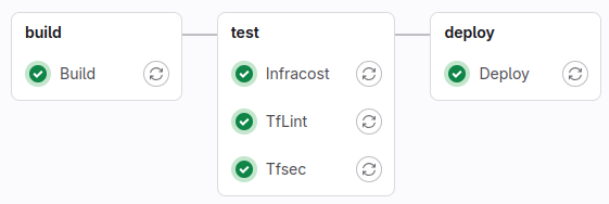

Figure 4: Deployment CI/CD pipeline.

expected verification based on the QoS requirements initially expressed by the FSA via the respective *QoS Solutions*. Finally, the "deploy" stage is triggered once the "test" stage has been successfully completed. As a result, the Fog system described in the selected *Deployment Configuration* file is actually deployed by relying on the corresponding deployment tool.

5 Practical Applications in Different Domains and for Different QoS Requirements (RQ3)

To validate the proposed approach (cf. Section 3) and related automation support (cf. Section 4), we evaluated VeriFogOps in the context of various examples of Fog systems belonging to different application domains. To this end, we considered two distinct use cases: *Smart Campus* [1] and *Smart Parking Lot* [7]. To illustrate the generic aspect of our approach, we experimented with two different types of QoS requirements which are particularly relevant in the context of Fog systems, namely security and cost. Table 1 shows the corresponding pairs *Deployment Tool – QoS Solution*.

In the following, we provide for each use case 1) a summary of the general context, 2) the result of the deployment tool selection, and 3) the result of the automated deployment based on previous selection. The full implementation of our approach, associated automation support, and application to these use cases is available in an open source repository (cf. Section 6 for more details).

5.1 Smart Campus with No Priority

Fog system. The first use case is taken from a recent survey of the state-of-the-art in terms of existing Fog Modeling Languages [1]. In this survey, Smart Campus is used as a motivating example introducing the main elements of a Fog system. This particular Fog system is mainly composed of two distinct distributed applications dedicated to smart surveillance and smart bell notification. These applications are made available as loosely coupled micro-services which are natively designed to be mutually shared.

Deployment tool selection. For this use case, we take as initial inputs two different *Deployment Configuration* files associated with two *Deployment Tools*: Terraform and Kubernetes. As introduced before, we considered two important QoS requirements which are security and deployment cost. Figure 5 shows the corresponding *Deployment Instructions* file as specified by the FSA.

Then, in the *Deployment Solutions Evaluation* file, the DevOps Engineer associates 1) Terraform with Kics (for security) and Infracost (for cost) and 2) Kubernetes with Kics (for security) and Kubecost (for cost). Based on all these inputs, the *Selection CI/CD Pipeline* compares the security results of each deployment configuration file against the thresholds specified in this *Deployment Instructions*

Table 1: Applications of the VeriFogOps approach.

Use Case Name	Deployment tool	Verified Selection Property	QoS Solution	Return Type
Smart Campus, Smart Parking	Terraform	Cost	Infracost	Table
	Kubernetes	Cost	Kubecost	Table
	Terraform & Kubernetes	Security	Kics	List + Table

```
tests:
  - name: ConfigurationSyntax
    rules:
      MAX_LINT_ISSUES: 5
    status: Blocking

  - name: Security
    rules:
      CRITICAL_PROBLEMS: 20
      HIGH_PROBLEMS: 40
      MEDIUM_PROBLEMS: 34
      LOW_PROBLEMS: 15
    status: Optional

  - name: Cost
    rules:
      BUDGET: 250
    status: Optional
```

Figure 5: Excerpt of the Deployment Instructions file for the Smart campus system.

files, as well as the estimated costs in order to select the least expensive solution. As no priority is indicated within the *Deployment Instructions* file, the pipeline selects the *Deployment Configuration* having the minimum number of critical security issues under the thresholds and also being the cheapest one in terms of deployment cost. Note that, in case the above evaluation step returns several possible results, the pipeline consider by default as a priority the first QoS requirement mentioned in the *Deployment Instructions* file. The results of the selection are shown in Table 2.

Table 2: Results of the Deployment tool selection for the Smart Campus system.

Deployment tool (configuration file)	QoS Solution	Results
Terraform	Kics	Critical issues 0
		High issues 16
	Infracost	201.56$
Kubernetes	Kics	Critical issues 0
		High issues 30
	Kubecost	261.69$

In this use case, Terraform, combined with Kics and Infracost, emerged as the most suitable option with only 16 high security issues and a cost of 202 USD. The output provided by our *Deployment Tool Selection* component is shown in Figure 6.

Automated Deployment. Given that Terraform is the most suitable tool for this scenario, the corresponding *Deployment Configuration* file is used as the input for the *Infrastructure Deployment* component. We assume that the original *Deployment Instructions* file also includes configuration syntax as a blocking requirement, with cost and security treated as optional requirements. In this use case,

```
Both critical values are 0, we analysed the high category
There is no priority found between cost and security
Terraform has the smallest high issues = 16
Terraform is the cheapest one, it costs 201.56 USD
$ echo "Selected choice is: $selected_choice"
Selected choice is: Terraform
```

Figure 6: Excerpt of the result of the Deployment Tool selection for the Smart Campus system.

the *Domain-oriented CI/CD Generator* produces a *Domain-oriented CI/CD* file for the Gitlab platform. This file integrates Infracost for cost estimation, Tfsec for security detection, and Tflint for configuration syntax checking. Note that we initially used Kics during the selection process due to it being compatible with both Terraform and Kubernetes. However, the SysOps Engineer can decide to use Tfsec instead of Kics in this stage as it offers more detailed and specific detection capabilities tailored for Terraform configurations. We considered in the updated *Deployment Instructions* file a budget of 250 dollars and a maximum allowable lint issues threshold of 5.

Figure 7 gives a partial example of the obtained results in terms of cost. Notably, 46 Cloud resources have been detected and the estimated deployment cost for 20 of them is displayed. For each one of these 20 resources, the instance usage per month, the storage that will be used, and the monthly cost are provided. The other detected Cloud resources appear to be free of charge, and are simply ignored when computing the total monthly deployment cost for the whole project. Note that 10 syntax issues were detected (i.e., more than the maximum allowed number of 5) thus blocking the pipeline execution. As a consequence, the SysOps Engineer needs to decide whether to modify the *Deployment Instructions* file (e.g., to allow for 10 syntax issues or more) or directly the *Deployment Configuration* file in Terraform following the provided syntax recommendations. Once the problem solved, the pipeline execution can continue until the actual deployment using Terraform.

5.2 Smart Parking Lot with Cost Priority

Fog System. The second use case is a solution for drivers to be more efficient when looking for a slot for their vehicles in a parking lot [7]. The solution describes an example of a Fog system which detects vehicles in each zone of the parking lot, and indicates available spaces to the drivers via a smart LED screen at the entrance. This system uses IoT devices to detect the presence of parked vehicles in each zone, and communicates the related data to the other components of the system in real-time.

Deployment tool selection. For the Smart Parking Lot use case, we followed the same process as for the Smart Campus use case. Thus, we took as initial inputs two different *Deployment Configuration* files associated with two *Deployment Tools*: Terraform

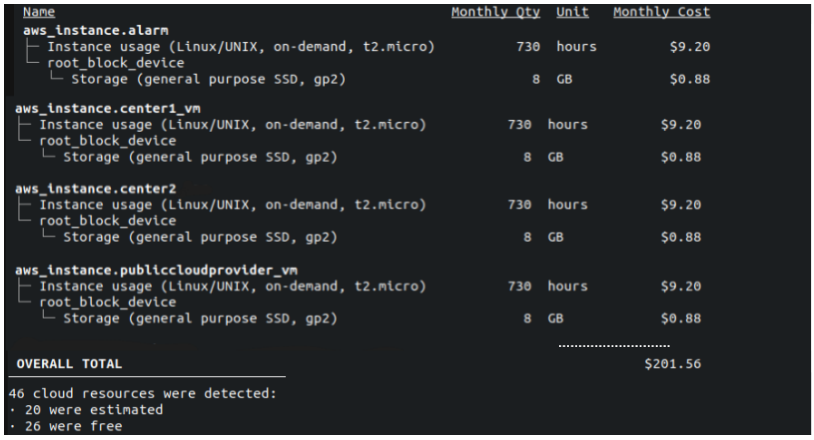

Figure 7: Partial example of results – Cost verification in Smart Campus with Infracost for Terraform.

and Kubernetes. We also considered two important QoS requirements which are security and deployment cost. Figure 8 shows the corresponding *Deployment Instructions* file as specified by the FSA.

```
tests:
  - name: Security
    rules:
      CRITICAL_PROBLEMS: 0
      HIGH_PROBLEMS: 17
      MEDIUM_PROBLEMS: 4
      LOW_PROBLEMS: 5
    status: Optional
    priority: 2

  - name: Cost
    rules:
      BUDGET: 200
    status: Blocking
    priority: 1
```

Figure 8: Excerpt of the Deployment Instructions file for the Smart Parking system.

Then, in the *Deployment Solutions Evaluation* file, the DevOps Engineer associates 1) Terraform with Kics (for security) and Infracost (for cost) and 2) Kubernetes with Kics (for security) and Kubecost (for cost). In this use case, we assume that the cost is the main QoS requirement, having the highest priority set compared to security. Following the same approach than for the Smart Campus use case, the results of the selection are shown in Table 3.

In this use case, Terraform, combined with Kics and Infracost, also emerged as the most suitable option. Although the Kics result for Kubernetes was also below the threshold, the pipeline selected Terraform as it is the cheapest option (232 USD). This is due to the priority given to the cost requirement compared to the security one. The output provided by our *Deployment Tool Selection* component is shown in Figure 9.

Automated Deployment. Given that Terraform is the most suitable tool for this scenario, the corresponding *Deployment Configuration* file is used as the input for the *Infrastructure Deployment* component. We assume that the original *Deployment Instructions* file includes a cost requirement as blocking (maximum budget of

Table 3: Results of the Deployment tool selection for the Smart Parking system.

Deployment tool (configuration file)	QoS Solution	Results
Terraform	Kics Infracost	Critical issues 0 High issues 46 231.79$
Kubernetes	Kics Kubecost	Critical issues 0 High issues 22 287.86$

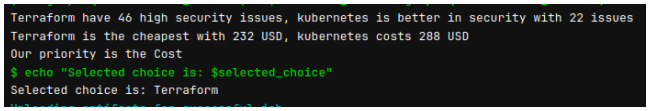

Figure 9: Excerpt of the result of the Deployment Tool selection for the Smart Parking system.

200 USD) and a security requirement as optional (maximum number of high-level problems of 17). Figure 10 shows the results obtained from the performed security verification. All the nodes were tested (i.e., no blocks were ignored), 23 nodes of the system did not have any security issues (i.e., 23 blocks passed) and 163 potential problems were detected: 46 are critical, 47 are highly risky, 1 is moderately risky and 69 are presenting a low risk. The number of detected security problems exceed the indicated threshold. However, as the security requirement is optional, it is not blocking the execution of the pipeline. Nevertheless, pipeline execution failed in this case due to a blocking cost verification. As a consequence, the SysOps Engineer needs to decide whether to modify the *Deployment Instructions* file (e.g., to augment the maximum budget) or directly the *Deployment Configuration* file in Terraform (i.e., to reduce the number of instances according to the verification indications collected before). Once the problem solved, the pipeline execution can proceed towards the actual deployment using Terraform.

```
Result #163 LOW Security group rule does not have a description.

  aws.tf:868-873

  859    resource "aws_security_group" "micro1_ins_security_group" {
  860      name = "micro1_ins_security_group"
  861      vpc_id = aws_vpc.edmm.id
  862      ingress {
  863        from_port = 22
  864        to_port = 22
  865        protocol = "tcp"
  866        cidr_blocks = ["0.0.0.0/0"]
  867      }
  ...

        ID aws-ec2-add-description-to-security-group-rule
    Impact Descriptions provide context for the firewall rule reasons
Resolution Add descriptions for all security groups rules

timings
-----------------------------------------
disk i/o            34.09µs
parsing             3.706177ms
adaptation          1.250562ms
checks              3.814931ms
total               8.80576ms

counts
-----------------------------------------
modules downloaded  0
modules processed   1
blocks processed    60
files read          1

results
-----------------------------------------
passed              23
ignored             0
critical            46
high                47
medium              1
low                 69

23 passed, 163 potential problem(s) detected.
```

Figure 10: Partial example of results – Security verification in Smart Parking with Tfsec for Terraform.

6 Implementation

Based on our own experience, technical expertise and previous work [5, 6], we decided to implement the proposed VeriFogOps approach within the Gitlab platform[7]. However, the approach itself is fully technology-independent, and could be implemented within any other suitable technical environment. For example, an alternative implementation could be developed by our partner company Smile by leveraging typical solutions such as Jenkins[8].

In the current implementation, we mostly considered Kubernetes and Terraform as *Deployment Tools*. Respectively, Dockerfile and Terraform YAMLs are used as the descriptive formats for the corresponding *Deployment Configuration* files. As presented earlier in Section 4, the language we proposed to specify the *Deployment Instructions* files also has a YAML syntax.

For the *Deployment Tool Selection* component, we implemented the CI file in GitlabCI[9] (another YAML notation) and used it in the Gitlab platform in order to execute the *Selection CI/CD Pipeline*. For the *Infrastructure Deployment* component, we implemented the *Domain-oriented CI/CD Generator* using several GitlabCI templates. The generator also integrates bash scripts combining these templates together based on the QoS requirements specified in the *Deployment Instructions* file. As a result it generates a GitlabCI file, i.e., the *Domain-oriented CI/CD* file. For example, in the context of the Terraform *Deployment Tool*, we implemented templates/support for Infracost (for cost requirements), Tfsec (for security requirements), and Tflint[10] (for configuration syntax requirements).

All the described resources are available in two open source repositories for the *Deployment Tool Selection*[11] and *Infrastructure Deployment*[12] components. This includes the implementation of these components in the Gitlab platform, examples of *Deployment Configuration* files for the use cases and *Deployment Tools* we mentioned, a corresponding example of a *Deployment Instructions* file, the complete source code of *Domain-oriented CI/CD Generator*, and an example generated Gitlab *Domain-oriented CI/CD* file.

7 Discussion

As mentioned before in Section 1 and Section 2, and later discussed in Section 8, the work presented in this paper leverages previously existing work regarding Fog systems, verification and automated deployment. However, its focus on QoS verification at deployment time makes it quite original compared to the current state-of-the-art. For example, we have not seen in the literature many solutions targeting explicitly the deployment tool selection phase. The idea of considering QoS requirements during such a selection phase, and even after for automating the actual deployment phase, is also original. We believe it is a relevant way of preventing from encountering unforeseen QoS-related problems after the deployment of the Fog systems. Finally, we argue that the automation of the overall process allows the engineers to gain some useful time they can possibly allocate to other important engineering activities.

The proposed VeriFogOps approach is generic because it supports the verification of different QoS requirements with different deployment tools and in the context of potentially any kinds of heterogeneous Fog systems (targeting various applications domains). This notably includes QoS requirements that are usually considered as critical in Fog systems, such as security and cost. Still, it is important to note that our intent is not to support by default in the current implementation of our approach all possible QoS requirements and deployment tools. This is the reason why we designed our approach as extensible so that 1) deployment instructions files can be refined in order to add the needed information regarding other QoS requirements than those mentioned in the paper and 2) additional deployment tools and QoS solutions to the ones considered in the paper, can be managed within the CI/CD pipelines whenever required by the engineers.

As a more global observation, genericity and extensibility (such as in the proposed approach) are key characteristics when addressing other phases of the Fog system's life cycle (complementary to the deployment time discussed in this paper). Automation is also a particularly important core aspect of VeriFogOps, as validated by our industrial partner Smile. Indeed, we provide both the automated selection of the most suitable deployment tool and the automated generation of domain-oriented CI/CD pipelines. Such pipelines can then be used in order to verify and ultimately deploy the targeted Fog systems. The work presented in this paper shows that providing such an automation support is actually feasible in

[7] https://about.gitlab.com/platform/
[8] https://www.jenkins.io
[9] https://docs.gitlab.com/ee/ci/
[10] https://github.com/terraform-linters/tflint
[11] https://gitlab.com/hiba.awad1/DDTS
[12] https://gitlab.com/hiba.awad1/TerraCI

practice. Moreover, it illustrates how current state-of-the-art tools (e.g., those mentioned in Section 2 and Section 6) can be concretely integrated to form a DevOps ecosystem really suitable for our industrial partner. To conclude concerning automation and integration, VeriFogOps provides the opportunity to leverage further, in other contexts and for various purposes, the different artifacts we select and/or generate (i.e., configuration files, CI files, CI/CD pipelines).

8 Related Work

8.1 Verification at Deployment Time

In modern distributed systems, such as Cloud and Fog systems, configurations are continuously deployed over time. In this context, misconfigurations [8] or configuration dependencies [24] are major sources of functional and reliability problems. As a consequence, solutions have already been proposed in order to detect such problems earlier, for instance at *pre-deployment time* through the use of model-checking or formal methods [11, 14, 22, 26, 27]. However, these solutions mostly focus on security-related issues and do not currently allow the verification of multiple QoS requirements. Other approaches such as SMADA-Fog [17] rely on techniques like simulation and linear optimization for deployment purposes. Still, they focus on specific goals (e.g., improving the system's overall performance) rather than on supporting various types of QoS requirements. With VeriFogOps, we intend to provide a more generic and extensible approach addressing different QoS requirements while possibly considering a prioritisation between them.

In parallel to these academic research efforts, Shift-left testing has recently attracted the attention of the DevOps community in industry (e.g., DevSecOps [18]). Shift-left testing is an agile software development strategy that allows engineers to discover their code problems (e.g., security issues) early in the engineering process. A study of Shift-left testing in DevOps [20] already identified its main benefits, challenges, and best practices in the context of DevOps environments. In a complementary way, another work analyzed the benefits of using a Shift-left solution for organizations [28]. The VeriFogOps approach described in this paper follows a similar path for verifying Fog systems at deployment time.

8.2 Automated Deployment

While traditional deployment activities most often require human intervention, the deployment of modern distributed systems, such as Fog systems, demands an increasing level of automation. As complexity, heterogeneity or dynamicity increase, approaches with underlying processes and tools are thus needed for supporting a more controlled and automated deployment [3].

For example, CloudCAMP [9] provided a generative programming approach integrated into an automated deployment and management platform for Cloud applications. As a result, it transforms partial specifications into deployable IaC code possibly targeting different Cloud providers. However, contrary to our approach, CloudCamp does not intend to support verification activities nor modern Fog systems. A more recent work proposed to rely on an automated CI/CD pipeline in order to deploy web applications [19]. Using a Gitlab file, the Docker Compose configuration of the web application is verified with three security tests detecting vulnerabilities. Then, the configuration can be deployed accordingly by the CI/CD in the Gitlab platform. However, unlike VeriFogOps, the proposed solution does not provide the possibility to parameterize these tests and only supports security concerns.

In another work [21], Azure DevOps is used to provide code management while developing and deploying the target system through a CI/CD pipeline for the Microsoft platform. This solution supports automated testing and validation using the Eggplant testing tool in order to ensure code quality. However, the provided pre-deployment process requires users to manually approve or reject deployments. In our approach, this is fully automated thanks to the expression of priorities on the expressed QoS requirements. Moreover, the proposed solution focuses only on code quality while our approach can also be used to verify other QoS requirements.

Overall, the previously existing solutions are not intended to support the verification of different types of QoS requirements, more particularly at deployment time. In addition, they do not natively target Fog systems, and do not come with a selection step concerning the most relevant deployment tool and corresponding deployment configuration file to be used. Despite the use of automated CI/CD pipelines in some cases, these solutions do not offer neither the possibility to parameterize the QoS requirements to be verified. The main objective of the VeriFogOps approach we propose in this paper is to make a step in this direction.

9 Conclusion and Future Work

In this paper, we proposed VeriFogOps as an automated deployment tool selection and CI/CD pipeline generation approach for verifying Fog systems at deployment time. This approach is intended to facilitate the engineering activities of both Fog system architects and DevOps/SysOps engineers. In practice, we provide a generic and automated approach for selecting the most suitable deployment tool and related deployment configuration file, while ensuring that the QoS requirements expressed by the architects can be satisfied. Once the deployment tool and deployment configuration file are selected, we provide a CI/CD generator that automatically produces a dedicated CI/CD pipeline in order to deploy the target Fog system. Interestingly, this pipeline also ensures that the previously expressed QoS requirements are automatically verified before the actual deployment of the Fog system. This is achieved by 1) combining deployment tools with QoS solutions in order to perform the QoS requirements verification that guides the selection of the most suitable deployment tool and related deployment configuration files, and 2) considering deployment instructions that incorporate the expressed QoS requirements into an automatically generated CI/CD pipeline that then performs the actual deployment of the target Fog system. We already applied our approach and its current implementation in practice in the context of two different use cases. As a result, we had to consider Terraform and Kubernetes as deployment tools, combine them with Kics, Infracost and Kubecost as QoS solutions, and generate CI/CD pipelines for the Gitlab platform.

In next steps, we plan to continue generalizing the use of our approach and its main principles in different phases of the Fog system's life cycle. For example, still at deployment time, VeriFogOps could be extended to support a more optimal resource management by also considering allocation and/or provisioning requirements (among others). Moreover, at execution time, VeriFogOps could be

completed in order to be used for dealing with the self-adaptation (e.g., reconfiguration) of the target Fog systems. Overall, this would allow the FSA and the DevOps/SysOps engineers to build more manageable, resilient and evolutive Fog systems.

On the longer term, an important objective is the integration of VeriFogOps and the generated CI/CD pipelines into real-time monitoring DevOps CI/CD pipelines that operate post-deployment (i.e., at execution time). The resulting "augmented" pipelines will notably have to support the building, maintenance, and monitoring of the deployed Fog systems in a more continuous and automated manner. From an industrial perspective, and as particularly interesting for our industrial partner Smile, continuous verification appears to be a promising path towards the iterative improvement of the QoS of Fog systems during their whole life cycle. We believe that generic and automated approaches such as VeriFogOps can lead to significant time, energy, and cost savings when engineering large and complex Fog systems.

Acknowledgment

This work was funded by the French Agence Nationale de la Recherche Technologique (ANRT) under a Cifre PhD grant and by the French Agence Nationale de la Recherche (ANR) under grant ANR-20-CE25-0017 (SeMaFoR project).

References

[1] Abdelghani Alidra, Hugo Bruneliere, and Thomas Ledoux. 2023. A Feature-based Survey of Fog Modeling Languages. *Future Generation Computer Systems* 138 (2023), 104–119.

[2] SAIBS Arachchi and Indika Perera. 2018. Continuous Integration and Continuous Delivery Pipeline Automation for Agile Software Project Management. In *2018 Moratuwa Engineering Research Conference (MERCon)*. IEEE, Piscataway, New Jersey, United States, 156–161.

[3] Jean-Paul Arcangeli, Raja Boujbel, and Sébastien Leriche. 2015. Automatic deployment of distributed software systems: Definitions and state of the art. *Journal of Systems and Software* 103 (2015), 198–218.

[4] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. 2010. A View of Cloud Computing. *Communications of the ACM* 53, 4 (2010), 50–58.

[5] Hiba Awad, Abdelghani Alidra, Hugo Bruneliere, Thomas Ledoux, Etienne Leclerq, and Jonathan Rivalan. 2024. VeriFog: A Generic Model-based Approach for Verifying Fog Systems at Design Time. In *The 39th ACM/SIGAPP Symposium on Applied Computing (SAC'24)*. ACM, New York, NY, USA, 1252–1261.

[6] Hiba Awad, Abdelghani Alidra, Hugo Bruneliere, Thomas Ledoux, and Jonathan Rivalan. 2024. VeriFog: A Generic Model-based Approach for Verifying Fog Systems at Design Time and Generating Deployment Configurations. *SIGAPP Applied Computing Review* 24, 3 (2024), 18–36.

[7] Kamran Sattar Awaisi, Assad Abbas, Mahdi Zareei, Hasan Ali Khattak, Muhammad Usman Shahid Khan, Mazhar Ali, Ikram Ud Din, and Sajid Shah. 2019. Towards a Fog-enabled Efficient Car Parking Architecture. *IEEE Access* 7 (2019), 159100–159111.

[8] Salman Baset, Sahil Suneja, Nilton Bila, Ozan Tuncer, and Canturk Isci. 2017. Usable declarative configuration specification and validation for applications, systems, and cloud. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Industrial Track* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 29–35.

[9] Anirban Bhattacharjee, Yogesh Barve, Aniruddha Gokhale, and Takayuki Kuroda. 2018. CloudCAMP: Automating the Deployment and Management of Cloud Services. In *2018 IEEE International Conference on Services Computing (SCC)*. IEEE, Piscataway, New Jersey, United States, 237–240.

[10] Antonio Brogi and Stefano Forti. 2017. QoS-aware deployment of IoT applications through the fog. *IEEE internet of Things Journal* 4, 5 (2017), 1185–1192.

[11] Claudia Cauli, Meng Li, Nir Piterman, and Oksana Tkachuk. 2021. Pre-deployment Security Assessment for Cloud Services Through Semantic Reasoning. In *Computer Aided Verification*, Alexandra Silva and K. Rustan M. Leino (Eds.). Springer International Publishing, Cham, 767–780.

[12] Christof Ebert, Gorka Gallardo, Josune Hernantes, and Nicolas Serrano. 2016. DevOps. *IEEE Software* 33, 3 (2016), 94–100.

[13] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K. Ghosh, and Rajkumar Buyya. 2017. iFogSim: A toolkit for modeling and simulation of resource management techniques in the Internet of Things, Edge and Fog computing environments. *Software: Practice and Experience* 47, 9 (2017), 1275–1296.

[14] Petar Kochovski, Pavel D. Drobintsev, and Vlado Stankovski. 2019. Formal Quality of Service assurances, ranking and verification of cloud deployment options with a probabilistic model checking method. *Information and Software Technology* 109 (2019), 14–25.

[15] Shancang Li, Li Da Xu, and Shanshan Zhao. 2015. The Internet of Things: a Survey. *Information systems frontiers* 17 (2015), 243–259.

[16] C. Mouradian, D. Naboulsi, S. Yangui, R. H. Glitho, M. J. Morrow, and P. A. Polakos. 2018. A Comprehensive Survey on Fog Computing: State-of-the-Art and Research Challenges. *IEEE Communications Surveys and Tutorials* 20, 1 (2018), 416–464.

[17] Nenad Petrovic and Milorad Tosic. 2020. SMADA-Fog: Semantic model driven approach to deployment and adaptivity in fog computing. *Simulation Modelling Practice and Theory* 101 (2020), 102033. Modeling and Simulation of Fog Computing.

[18] Roshan N Rajapakse, Mansooreh Zahedi, M Ali Babar, and Haifeng Shen. 2022. Challenges and Solutions when Adopting DevSecOps: A Systematic Review. *Information and software technology* 141 (2022), 106700.

[19] Thorsten Rangnau, Remco v. Buijtenen, Frank Fransen, and Fatih Turkmen. 2020. Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines. In *2020 IEEE 24th International Enterprise Distributed Object Computing Conference (EDOC)*. IEEE, Piscataway, New Jersey, United States, 145–154.

[20] V Shobha Rani, A Ramesh Babu, K Deepthi, and Vallem Ranadheer Reddy. 2023. Shift-Left Testing in DevOps: A Study of Benefits, Challenges, and Best Practices. In *2nd International Conference on Automation, Computing and Renewable Systems (ICACRS)*. IEEE, Piscataway, New Jersey, United States, 1675–1680.

[21] Farhana Sethi. 2020. Automating software code deployment using continuous integration and continuous delivery pipeline for business intelligence solutions. *International Journal of Innovation Scientific Research and Review* 2 (2020), 445–449. Issue 10.

[22] Ilia Shevrin and Oded Margalit. 2023. Detecting Multi-Step IAM Attacks in AWS Environments via Model Checking. In *32nd USENIX Security Symposium (USENIX Security 23)*. ACM, Anaheim, CA, 6025–6042.

[23] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. 2016. Edge Computing: Vision and challenges. *IEEE internet of things journal* 3, 5 (2016), 637–646.

[24] Sebastian Simon, Nicolai Ruckel, and Norbert Siegmund. 2023. CfgNet: A Framework for Tracking Equality-Based Configuration Dependencies Across a Software Project. *IEEE Transactions on Software Engineering* 49, 8 (2023), 3955–3971.

[25] Larry Smith. 2001. Shift-Left Testing. *Dr. Dobb's Journal* 26, 9 (Sept. 2001), 56, 62.

[26] Riza O. Suminto, Agung Laksono, Anang D. Satria, Thanh Do, and Haryadi S. Gunawi. 2015. Towards Pre-Deployment Detection of Performance Failures in Cloud Distributed Systems. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*. USENIX Association, Santa Clara, CA, 8.

[27] Xudong Sun, Runxiang Cheng, Jianyan Chen, Elaine Ang, Owolabi Legunsen, and Tianyin Xu. 2020. Testing Configuration Changes in Context to Prevent Production Failures. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. ACM, New York, New York, United States, 735–751.

[28] Srinivas Aditya Vaddadi, Ramya Thatikonda, Adithya Padthe, and Pandu Ranga Rao Arnepalli. 2023. Shift Left Testing Paradigm Process Implementation for Quality of Software Based on Fuzzy. *Soft Computing* Online (2023), 1–13.

[29] William Tichaona Vambe, Chii Chang, and Khulumani Sibanda. 2020. A Review of Quality of Service in Fog Computing for the Internet of Things. *International Journal of Fog Computing (IJFC)* 3, 1 (2020), 22–40.

[30] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. 2018. Fogernetes: Deployment and Management of Fog Computing Applications. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, Piscataway, New Jersey, United States, 1–7.

[31] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P. Jue. 2019. All One Needs to Know About Fog Computing and Related Edge Computing Paradigms: A Complete Survey. *Journal of Systems Architecture* 98 (2019), 289 – 330.

[32] Fiorella Zampetti, Salvatore Geremia, Gabriele Bavota, and Massimiliano Di Penta. 2021. CI/CD pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, Piscataway, New Jersey, United States, 471–482.

[33] Polona Štefanič, Matej Cigale, Andrew C. Jones, Louise Knight, and Ian Taylor. 2019. Support for full life cycle cloud-native application management: Dynamic TOSCA and SWITCH IDE. *Future Generation Computer Systems* 101 (2019), 975–982.