Ciências ULisboa

Phase 6 – Essay and Presentation

Grupo 2:

Diogo Lança – fc53495

Meuri Canhanga – fc42559

Miguel Silva – fc58974

Pedro Santos – fc46417

Renato Ribeiro – fc57433

1. Towards Continuous Consistency Checking of DevOps Artefacts

In this paper, the authors propose a Model-Driven Engineering (MDE) approach, where they represent each artefact as a model, establish links across them to set a navigable network of model elements and enable MDE services on top of the network.

It's explicitly quoted in the paper that "in MDE, artefacts are models, which are the cornerstones throughout the whole engineering lifecycle", and that "models are machine-readable, abstract representations of the designed (software) systems, enabling analysis and design techniques on a higher level of abstraction".

1.1. Approach

The approach is based on the generalization of the JSON-EMF Bridge – a tool-supported to bridge JSON schema technical space to Eclipse Modeling Framework (EMF) while preserving the native JSON concrete syntax.

For continuous consistency checking, the authors propose the adoption of GitOps practices, where all artefacts are within Git. The authors setted agents to detect changes in the artefacts and trigger the consistency checking service, which update the artefacts and push them back to the Git repository.

For evaluation, the authors used Keptn as a DevOps platform – it's an open-source tool for automation based on JSON schema. Keptn can automatically push the configuration files to Git. A Proof of Concept (PoC) was also implemented used for Quality Assurance (QA): Service Level Objectives (SLO) and Service Level Indicators (SLI). Theoretically, because of the PoC, its possible to transform the configurations of the SLO and SLI to models, check their consistency and apply basic model repairs, transforming them back into valid artefacts for Keptn.

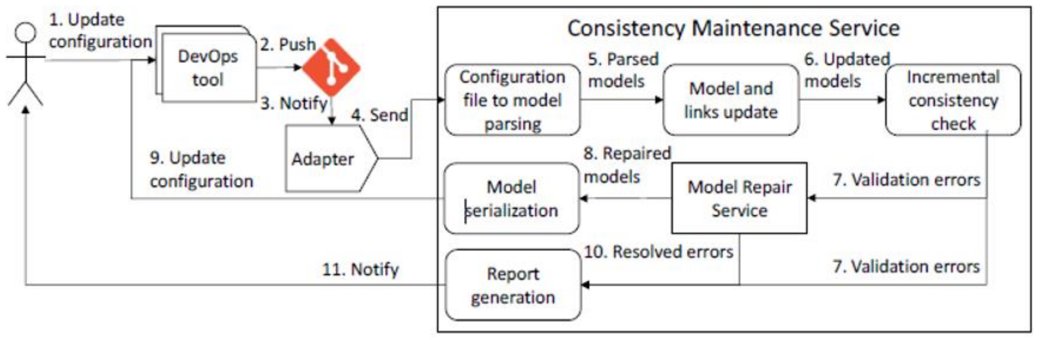


Figure 1– DevOps artefacts consistency management workflow

With a total of 11 steps to go through the workflow (fig.1) proposed by the authors, the objective of this approach, **if adapted to other technical spaces typically used in DevOps**, example - YAML, XML, Bash scripts, is for "every DevOps configuration conforming to an explicitly defined modeling language be parsed as a model while preserving the original format and compatibility with regard to the native DevOps tool". Resulting in a link between DevOps and EMF in practice.

1.1.1. Management workflow summary

In the first step of the figure 1, the DevOps engineer updates the configuration files of a selected DevOps tool, which automatically pushes the changes to a Git repository. The repository notifies the adapter, which, on the one hand, gets the added or deleted files and forwards them to the Consistency Maintenance Service (CMS).

In CMS, the artefacts are first parsed to EMF models, using the generalization of the process. The parsed models are compared with previous revisions to detect changes and update the models for the new revision. The linking model is also updated - model links are created or deleted if new models are added or removed, respectively. The consistency of the models is checked by executing validation rules by an incremental model query engine (viatra was used by the authors). The query engine is notified about the changes in the models and incrementally evaluates the rules. After that, the collected validation errors are forwarded to the Model Repair Service (MRS).

The MRS is responsible for restoring the model's consistency by applying quick fixes for simple cases as model transformations. The output of MRS consists of repaired models that are serialized using the native concrete syntax used by the considered DevOps artefacts (in this case JSON is the native concrete syntax).

At the end of the consistency management workflow, the validation errors and the corresponding fixes, if any, are collected in a report sent to the DevOps engineer, who is in charge of reviewing the workflow execution and possibly fixing manually those validation errors that were not automatically resolved.

1.2. Current Limitations

The current PoC proposed by the authors has the following limitations:

1. The consistency checking scenario is limited to two artefacts and a single DevOps tool.
2. The links in the linking model are established manually.
3. The consistency management workflow of fig. 1 has been implemented partially. It is limited to model update, incremental consistency check, model repair, and model serialization steps.
4. Propagation issue between MDE techniques. The DSL editor (from the EMF), generated with Xtext, does not forward change notifications, preventing the use of the incrementality feature by Viatra.
5. Scalability evaluation was not thought/added to the implementation on this PoC.

1.3. Conclusion

In this paper, the authors identify MDE techniques (EMF, Viatra, JSON-EMF Bridge), platforms (Keptn, GitOps) and use cases for DevOps configuration artefacts developed with language families. The proposed approach in the paper its still "faulty", where more research it's needed. The authors aim to implement the whole consistency workflow, where they plan to include more technical spaces besides JSON and will keep using the platforms selected for this proposal, meaning that for the time being they still don't consider the platform limitations as an issue.

2. An Advanced DevOps Environment for Microservice-based Applications

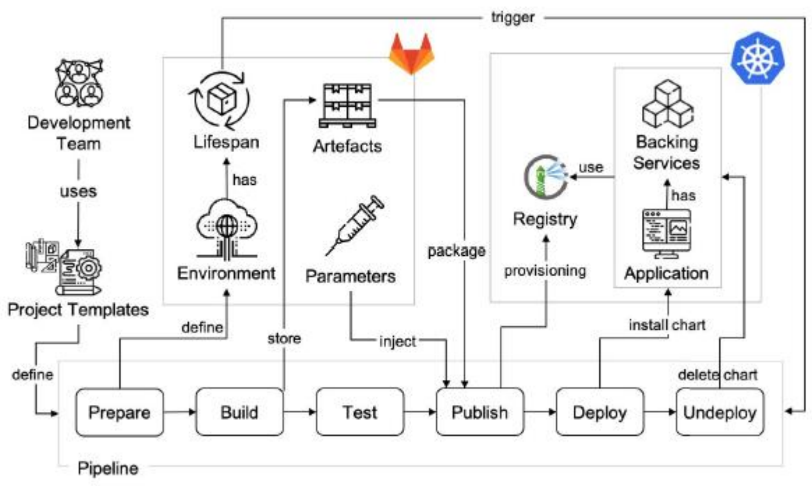
Over the last five years, microservices have been gaining more attention and nowadays the newest software projects are based on the microservice architecture, this provides several advantages compared with the monolithic approach. The main focus of this study is how to turn the integration of this microservice-based application with DevOps easy for developers. The approach on this study also aims for the security of all the services.

2.1. Template Based DevOps Environment

The template-based pipeline approach, aims to lower the entry barriers for developers to provide applications as well as reduce the maintenance efforts of the operational team. In the GitLab internal referencing mechanism single files of individual steps can be injected and shared across multiple projects. In this approach, each project template contains a predefined pipeline definition, which references the local project variables as well as shared pipeline templates.

The shared pipeline for Docker images includes build, push, and scan the images for known security vulnerabilities. The individual pipeline steps are then executed by a referenced pipeline image. The include mechanism in GitLab allows the user to reference the relevant files in other projects. Additionally a reference to a tag or branch of the file can be set. There were identified two types of pipeline steps: framework-independent and framework-specific.

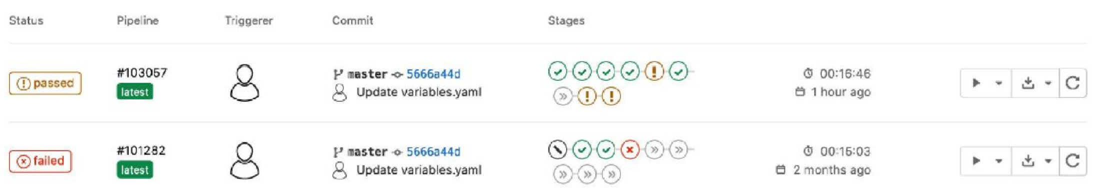
To prevent the changes to the shared pipeline templates from having a negative impact on the existing pipelines, a workflow for the maintenance of the shared pipeline templates was introduced. Each change to the pipeline templates or the deployment image triggers a different test to prevent any negative influences.



In the first step, certain rules are checked, and a linting of the project is performed. Internal guidelines and faulty configurations can be detected. The prepare step creates a GitLab environment that allows us to define a stop procedure and a lifespan for the deployment. This lifespan is limited to feature branches so as to save costs and still provide a test environment for other developers. All build artifacts are passed onto the next steps. Next, in the publish step, the Docker image is built, and the template of the Helm chart template is rendered and delivered to a Harbor registry. The delivered artifacts are then deployed to a predefined Kubernetes cluster.

2.2. Devoloper's Prespective

Templates are available for several kinds of projects, these include Go, Node, Java, Gradle, and Maven and assist developers with the specification of the DevOps environment without requiring them to know every detail of the underlying pipeline steps. Before a development team can start a new project, it must decide on a programming language to use, then a new Git project must be created, and the corresponding template must be imported. After this the project contains the basic structure with the necessary files for the application. It is also possible to integrate the shared pipelines in existing projects, this task requires a little bit more work and understanding of the workflow. When a developer pushes a commit to the repository, the pipeline is triggered, and they are presented with a detailed pipeline overview in GitLab. The developer can see the execution status of the pipeline and the currently executed stage. A detailed log is available for each step.



Status	Pipeline	Triggerer	Commit	Stages		
passed	#103057 latest		master 5666a44d Update variables.yaml		00:18:46 1 hour ago	
failed	#101282 latest		master 5666a44d Update variables.yaml		00:15:03 2 months ago	

Overall, the templates were positively received by the developers. Simple issues such as naming conventions in configuration files can lead to failures in the execution of the pipeline. Developers reported that the execution is quite complex and can be difficult to understand. These problems were fixed, and each project template now has comprehensive documentation.

2.3. Security Aspects

Establishing a high level of trust towards the pipeline is essential. So each step is executed in its own Docker container, and the containers are ephemeral, meaning that they are destroyed after completing the respective pipeline step. A dependency scan was also introduced to all existing pipelines by adding a new pipeline step definition to an existing template file. This centralized approach greatly reduces the amount of administrative effort required. Although a decomposed pipeline structure provides strong security advantages, it falls far short of solving all pipeline-related security issues. The main issues that have been identified are insufficient handling of credentials solved by using the GitLab environment variables made available to the pipeline containers, unrestricted outbound network access solved by modifying the GitLab pipeline implementation to tag each container with the name of the step that it executes. The crucial point is that network policies can be applied to specific tags, and thus the modified pipeline runner enables the implementation of least-privilege network access for pipeline containers. An important note regarding the above approach is that the internal DNS service of GitLab and Kubernetes have to be accessible by all the pipeline containers. Otherwise, the pipeline cannot be executed successfully.

The dependency scanning can help engineers to keep a security overview. An additional step vulnerability scan can be defined in the shared pipeline. This scan step is based on a Python script that interacts with the Harbor HTTP API. Once the result is available, the script creates a vulnerability report and finishes execution with a successful or unsuccessful exit code. The report is presented as a human-readable JSON file, which can be used by developers to fix any detected vulnerabilities.

2.4. Conclusion

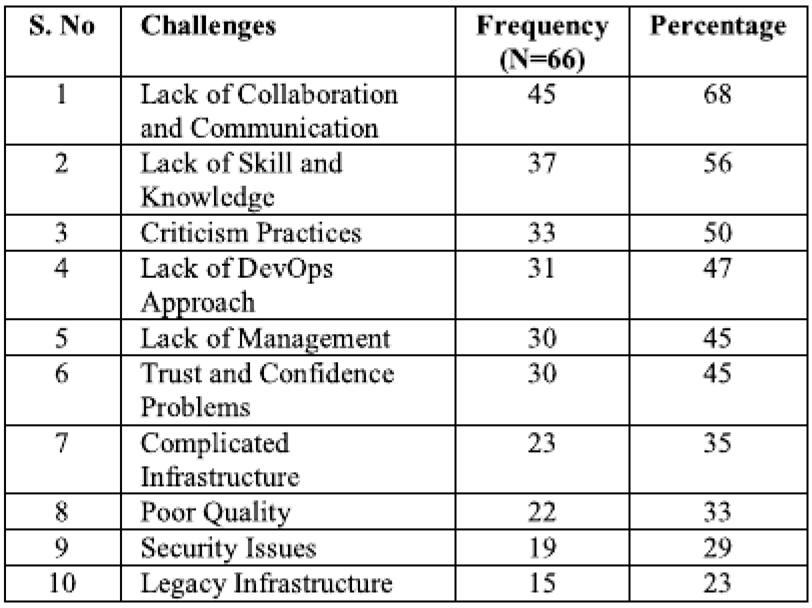
Through the DevOps workflow introduced above, was provided an environment that supports the deployment process of microservices and enforces security patterns in their implementation. Since most developers are not familiar with Docker or Kubernetes, the target was to make the introduction of these technologies and the integration of CI/CD concepts as simple as possible. This aim was achieved through the use of the project templates, freeing developers from the responsibility of creating the Docker or Kubernetes relevant artifacts and delivering them to the orchestration system. The template-based approach drastically reduced the maintenance times as well as the time to market for new microservice developments. To protect the pipeline and minimize attack vectors, the security-first pipeline structure was used.

3. Critical Challenges to adopt Devops Culture in Software Organizations: A Systematic Review

This paper discusses the DevOps approach to development, which is a method that combines the efforts of two different teams: Developers and Operations teams. The advantages to this sort of approach are immediately clear upon experimentation: There's a shorter time to production and all the versions of software produced through this method tend to be more stable and reliable than if they'd been produced through more traditional methods. These significant advantages make DevOps very appealing as a software development and deployment method which is why it is nowadays so widespread and accepted. Unfortunately, the lack of understanding of key concepts, good practices tools and challenges related to this approach make its adoption significantly harder. This paper thus attempts to mitigate this issue by first discovering and then exploring and discussing the challenges related to DevOps culture and practices.

3.1. Methodology

The authors of this paper used an SLR protocol to search through 11495 papers and articles regarding DevOps in order to identify the problems. The databases searched were IEEE Xplore, ACM Digital Library, Springer Link, Google Scholar and Science Direct They claim the specific protocol they used will be published at a later date, but from this particular paper we can gather that they first constructed an initial search string (DevOps AND Culture AND Challenges AND vendor) and then expanded upon it. The next step was to introduce some inclusion and exclusion criteria which reduced the targeted papers number to 380. Finally after manually reading through those, the authors ended up with 66 papers and articles that fit their criteria. Upon examining those, the authors produced a table with the 10 most critical challenges found and the frequence they appeared with within these papers in both numerical and percentage form.



S. No	Challenges	Frequency (N=66)	Percentage
1	Lack of Collaboration and Communication	45	68
2	Lack of Skill and Knowledge	37	56
3	Criticism Practices	33	50
4	Lack of DevOps Approach	31	47
5	Lack of Management	30	45
6	Trust and Confidence Problems	30	45
7	Complicated Infrastructure	23	35
8	Poor Quality	22	33
9	Security Issues	19	29
10	Legacy Infrastructure	15	23

3.2. The 10 Critical Challenges in DevOps Environment

The most critical challenge found was the lack of collaboration and communication between the different teams. In 68% of the papers considered, this issue was found. The problem seems to be that Developers and other IT operations teams do not share their common goals and plans, which turns appropriate communication into a very difficult goal to achieve and has as consequence software delays. These people with distinct professional backgrounds need to be properly managed otherwise they lose sight of their goals.

In 56% of the papers the lack of skill and technical knowledge was identified as an issue, making it also a significant challenge. The fact is that properly implementing DevOps practices requires both development and operational knowledge and companies simply do not have many skilled workers who are skilled in both areas. The results in professionals lacking a knowledge of key concepts, methods, tooling and key benefits and challenges to implementing DevOps, which is a problem that isn't helped by the fact that most people are only interested in their area of expertise.

Then we have Criticism practices with a 50% frequency. This represents a blaming culture that harms the DevOps culture because it leads to destructive behavior when the key to success here is cooperation between teams.

At 47% we have a lack of DevOps approaches which is essentially development and IT teams working separately by standard and thus not having much if any experience in the DevOps environments. This issue might seem obvious but it's amplified by the fact that both teams rarely if ever actually cooperate directly on a daily basis.

Lack of Management is the next issue at 45% frequency, and it represents improper management by team leaders when presented with team members that are facing a fundamental change to their jobs. This issue makes a nasty pair with trust and confidence problems (45%) which happens when people work in an environment of distrust between teams and fear of losing their jobs.

The use of many tools for DevOps represents the complicated infrastructure issue (35%) which is an adversity for the implementation of DevOps. Poor cooperation in turn leads to poor quality (33%) software which may fall short of the standards set by the company because the operations department might lack confidence in putting software in production which can delay software and in turn delay fixes of things such as security issues.

Security in itself is a barrier found with 29% frequency, with the major problem being allowing people access when it's needed, but restraining it when it's not, as well as not considering security procedures in advance. Finally, legacy infrastructure (23%) seems to also be a common issue as organizations tend to stick to their traditional practices and then lack the expertise in methods of new frameworks which makes it difficult to persuade firms of the long-term benefits of this new DevOps approach.

3.3. Conclusion

This study has identified a list of 10 Critical Challenges, six of which were considered pressing matters that need to be addressed as soon as possible and with full attention in order to successfully adopt a good DevOps approach in a firm, company or work group. These six challenges are: 'Lack of collaboration and Communication', 'Lack of Skill and Knowledge', 'Complicated Infrastructure', 'Lack of Management', 'Lack of DevOps approach' and 'Trust and Confidence Problems'.

4. Software Reliability in a DevOps Continuous Integration Environment

Since manually tracking software reliability is a labor intensive task, the paper explores the feasibility of obtaining reliability metrics autonomously by integrating into an existing DevOps/Continuous Deployment (CD) environment pipeline.

Reliability can be defined as a prediction or estimated measure of the failure free performance of software over a specific interval under specific conditions. As a prediction, reliability is a tool to anticipate the resources needed to discover and resolve defects throughout the software development lifecycle. As a risk reduction measure, coupled with an initial prediction, reliability is an instantaneous gauge of the efficiency of the software development process in achieving failure free performance over time.

4.1. Plan of Action

In order to form a path forward on implementation of software reliability in a DevOps Continuous Integration Environment understanding of the current state is key. The paper authors describe a plan of action comprised of the following steps:

- Review documentation on Software reliability
- Document definitions and types of data needed for software reliability models.
- Work to find alignment between critical DevOps metrics, company software reliability guidelines and the IEEE 1633
- Research current programs using DevOps tools across the company, gather their names and study lessons learned.
- Conduct a survey to gather data about the current state of software reliability across the company.
- Compile results from the survey into a master list.
- Create a list of manual steps needed to have a robust automated software reliability program.
- Create a matrix providing the relationships between base data gathered in a DevOps/ Continuous Integration (CI) environment, metrics, stakeholders, and tools.

4.2. RelationShip Matrix

The paper authors developed a relationship matrix which details the base data needed for most software reliability models, a list of metrics aligned to measure key DevOps outcomes along with software reliability, the stakeholders involved, and the tools that are currently used. The matrix identified 4 areas of automation related to tools which would give us access to 13 base datas measures tied to 6 metrics. The body of metrics and data would provide a way to fulfill any software reliability/maturity requirements along with the foundations needed for modeling reliability or reliability growth.

There are no industry standards on software reliability in a DevOps /CI environment. The paper suggests that the individual program leadership understand the current definitions pertaining to software reliability in general, such as:

Defect: A problem that, if not corrected, could cause an application to either fail or to produce incorrect results. it's the result of errors that are manifest in software requirements, architecture or code…

Feature: A feature is a unit of functionality of a software system that satisfies a requirement.

LRU: A software Line Replaceable Unit (LRU) is the lowest level of architecture for which the software can be compiled and object code generated.

Stress testing from Critical Items List: Generate a critical items list (CIL)—The output of the software failure modes and effects analysis (SFMEA) is the list of critical items and their associated mitigations

CSCI: Computer Software Configuration Items (CSCI) will be included. Generally, the analysis should cover all CSCIs that can affect system critical mission performance or personnel safety.

Legend
Automated
Easy to Automate
More difficult to Automate
Most Difficult to Automate
Metrics of Interest
Metrics of not in the Scope of th project

	Base Data - measure - one value no equation													**Stakeholders**					
Metrics	Actual KSLOCS	Actual Faults/Defects/Errors Autonomous	SW Version Number or timestamp	# of Software requirements	Test Completeness/Effectiveness	# Software LRUs/ CSCI/ Features	# Unit Tests Failures	# Verification Test Failures	Test Execution Start/End Timestamp	Software LRUs/ CSCI/ Features TimeStamp	Fault/Failure Discovery Timestamp	Fault Recovery Timestamp	Actual Complexity	Program Reliability Lead	Reliability Discipline Owner	Program Manager	Customer	Software development metrics owner	Process Group
Defects in Ops (near-Ops) per Deploy (DevOps)	x	x	x											x	x	x	x	x	x
Automated Test Requirements Coverage	x			x	x									x	x	x	x		x
Structural Code Coverage					x	x								x	x	x	x	x	
Pass/Fail Rates				x	x	x	x	x						x	x	x	x		x
Stress test coverage- testing critical faults					x			x	x					x	x	x	x		x
Integration Test results and coverage					x		x	x	x					x	x	x	x		x
Mean Time to Recover											x	x		x	x				
Feature Cycle Time						x				x				x	x				
Tools (sources of Base Data)																			
LCT/Sizer	🟩																		
Project Tracking Tool (**Jira, TFS**, RTC, iTracker, …)		🟩	🟩							🟩	🟩	🟩							
Requirements Tracking (**Doors, DNG**…)				🟩															
Continous Integration Environment **Jenkins**			🟨		🟩	🟩	🟩	🟩	🟩		🟩	🟩							
Static Analysis (Coverity, SonarQube, Clang-tidy, Find Bugs, CheckStyle, PMD, …)		🟨											🟩						
Unit Test (CPP Unit, Google Test, Boost Test, Junit, Py Nose2, …)		🟨					🟩		🟩		🟩								
Software Verification Test (Cucumber, Eggplant, Squish, …)		🟨						🟩	🟩		🟩	🟩							
Structural Coverage (Bullseye, gcov, jcov, Clover, Covertura, …)					🟩														
Code Review Tool (ReviewBoard, Gerrit, Collaborator, …)		🟨																	
Autonomous Deployment (Artifactory, Nexus, …)			🟨																

4.3. Conclusion

The base data exists in the DevOps/CI environments, and its collection can be automated to reduce the man hours needed to estimate software reliability. By automating the collection and reporting of reliability data within a DevOps pipeline would provide the following benefits:

- Instantaneous visibility into the effectiveness of software verification
- Gauge of software maturity at any point in development cycle
- Facilitates dynamic recalibration of resource scheduling
- Minimize impact to developers while providing feedback about their progress
- Provide dashboards to allow real-time comparisons between predicted vs estimated reliability

5. Monitoring solution for cloud-native DevSecOps

Software development and operations are increasingly adopting cloud-native environments. The popularity of development practices such as DevSecOps is one of the reasons for this change. It is identified that monitoring is one essential practice in DevSecOps and currently, a wide variety of tool offerings are available on the market to address this new transformation. However, an automated monitoring solution that covers both the infrastructure and application level is not available yet. In this paper, the authors have proposed an automated monitoring solution for the cloud-native DevSecOps.

5.1. Overview of the solution

DevSecOps teams usually consist of three groups: Development, Operations and Security teams and they all interact with the cloud-native infrastructure.

The monitor system monitors two aspects: Infrastructure monitoring (cloud monitoring) and Application monitoring.

The infrastructure monitoring service monitors the general status of the different deployed environments. This is vital for the three teams (Development, Operations and Security) in DevSecOps since they all work in the same environment. The infrastructure monitoring provides four values based on the time: System health (System is online or not), CPU Rate (The average CPU usage), File system usage (usage of active disk space) and Network Usage (The network activity).

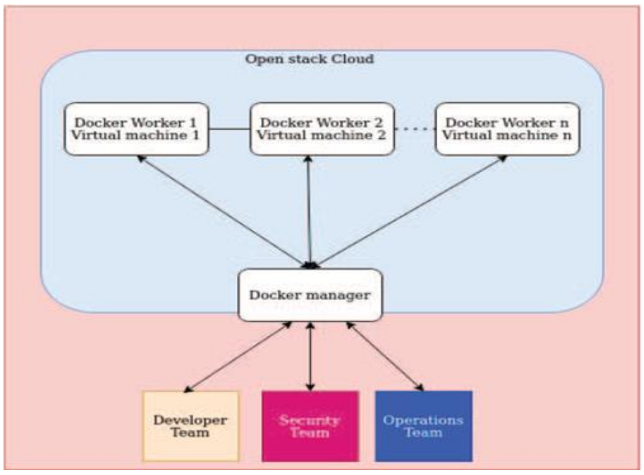
The application monitoring monitors the continuous integration and continuous deployment pipeline. In addition, for security monitoring the authors have included an open-source package that can identify the location information of the users from the IP address.

5.2. Components of the monitoring solution

One of the key practices in DevSecOps is automation where the need for specialized tools is high. The authors' proposed solution is primarily aimed at the operations teams after the deployment phase with a focus on security. In the future, the authors plan to extend this method to the development teams.

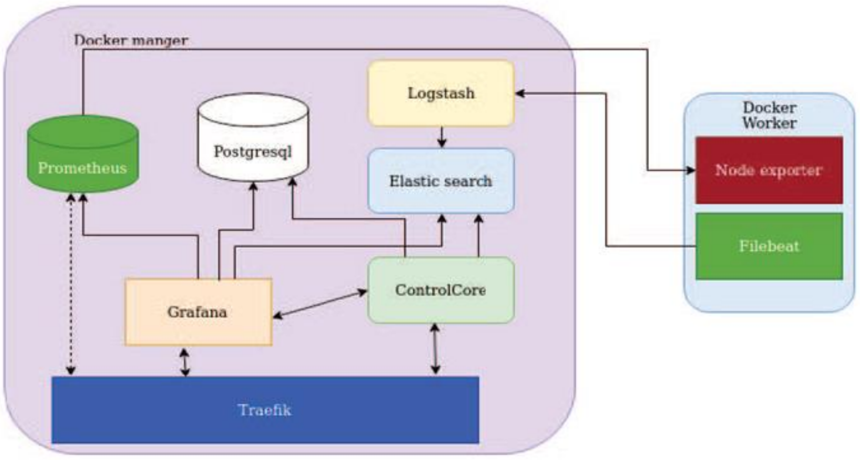
For this paper, the authors used an open-source cloud solution called **OpenStack**, for Infrastructure as a Service (IaaS).

The authors used a Microservice Architecture Style (MSA) based architecture using Docker to ensure scalability and easier control of the solution:


Open stack Cloud
Docker Worker 1 Virtual machine 1
Docker Worker 2 Virtual machine 2
Docker Worker n Virtual machine n
Docker manager
Developer Team
Security Team
Operations Team

The Docker manager is the main Docker container that has all the critical components, and all the other machines that need monitoring will act as Docker worker nodes.

This figure shows the different components deployed using Docker and their interfaces in the monitoring solution:



- Elasticsearch and Logstash – Used for collecting the application level log information.
- Filebeat – Data shipper to collect information from various sources.
- Traefik – Supports routing, reverse proxy and load balancing.
- Grafana – Used for visualization. Uses Elasticsearch API.
- Node Exporter – Collector agent that collects system-level information.
- Prometheus – Time Series Database (TSDB).
- PostgreSQL – Stores any persistent data for the application.

In this solution proposed by the authors, it was used agent-based monitoring and the data collection is based on the agents running on the worker node. The worker node has two agents, **Node Exporter** and **Filebeat**. The Node Explorer exposes the system level monitoring information (CPU usage rate, memory usage) and Filebeat collects the data from application logs. The information from Node Exporter is stored in **Prometheus**. This stored information is constantly retrieved by **Grafana** and stored in its database for visualization. The data exposed by FileBeat is passed through **Logstash** and stored in **Elasticsearch**. The data from Elasticsearch is retrieved by Grafana to populate the application level information. Finally, all the external connections are handled by **Traefik**. The main advantage of using Traefik was security, in addition, Traefik also solves the problem of port allocation in Docker, which was one of the **challenges** for the authors.

5.3. Conclusion

Cloud-native DevSecOps monitoring is challenging. As it was said at the beginning of the paper by the authors, although there are some tools and practices for cloud-native DevSecOps, they only focus on some of the aspects of monitoring. This results in a lack of comprehensive DevSecOps monitoring solutions.

This paper addressed this gap by developing a novel monitoring solution for infrastructure and application as an integrated solution, although the authors said that it has several limitations. For example, the authors didn't present an evaluation based on the proposed solution. In the future, the authors intend to address this limitation by doing a detailed evaluation based on an experimental case.