An Advanced DevOps Environment for Microservice-based Applications

Stefan Throner, Heiko Hütter
iC Consult Group
Huyssenallee, 45128 Essen, Germany
(stefan.throner | heiko.huetter)@icconsult.com

Niklas Sänger, Michael Schneider, Simon Hanselmann, Patrick Petrovic, Sebastian Abeck
Research Group Cooperation & Management
Karlsruhe Institute of Technology (KIT)
Zirkel 2, 76131 Karlsruhe, Germany
(niklas.saenger | sebastian.abeck | michael.schneider)@kit.edu

2021 IEEE International Conference on Service-Oriented System Engineering (SOSE) | 978-1-6654-3477-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/SOSE52839.2021.00020

***Abstract*—Complex applications consisting of many interdependent microservices require an advanced environment that allows their efficient Development and Operations (DevOps). One of the central components of a DevOps environment is a pipeline concept that supports the Continuous Integration/ Continuous Deployment (CI/CD) of single microservices, usually in the form of a container-virtualized cloud infrastructure based on advanced technologies such as Docker, Kubernetes, or Helm. Although there are available concepts and technologies to implement these concepts, it remains unclear how to combine the concepts and technologies into an advanced DevOps environment which specifically supports the different roles involved in the process. This paper describes the DevOps environment set up to develop microservice-based applications and focuses on the following aspects: (i) a flexible CI/CD pipeline based on reusable templates, (ii) support for developers to use the DevOps environment efficiently, and (iii) the security of the environment against attacks.**

1. Introduction

Over the last five years, microservices [1] have been gaining more widespread attention. Today, many newly developed software systems are based on the microservice architecture, and existing monolithic systems are divided into microservices [2], which provide several advantages. One advantage is that the microservice architecture is based on a domain-driven design [3] which results in well-defined microservice APIs [4] that can be reused in many applications. This may improve the maintainability of the software system, especially compared to a monolithic system. A second advantage is that since each microservice can be built, tested, and deployed on its own, such applications support the concept of continuous Integration/Continuous Deployment (CI/CD) [5]. In a professional environment, complex applications with many interdependent microservices are deployed, and this process must be supported by a powerful CI/CD pipeline as part of a DevOps setup. A third advantage of microservices concerns their operational aspect. As the virtual computing unit of the architecture, a microservice fits a container-virtualized infrastructure which uses technologies such as Docker and Kubernetes. Such infrastructures support both vertical and horizontal scaling, which result in flexible usage of the available computing resources by the running microservice-based applications [6].

For developers, the additional abstraction and tools require additional knowledge and work, resulting in a shift in focus from software development to additional operational overhead [7]. The DevOps environment introduced in our approach can drastically reduce this overhead and enforce quality and security standards for all services. Furthermore, our approach minimizes the maintenance effort and infrastructure costs without compromising the developer's experience. To demonstrate its effectiveness, we tested our approach using a project with multiple microservices and validated the results.

Figure 1 illustrates the central elements of the DevOps environment that we have configured to build, test, deploy, and run microservice-based applications from various domains. The following chapters take two such applications, PredictiveCarMaintenance and ClinicsAssetManagement (from the connected car and healthcare domains, respectively) as examples.

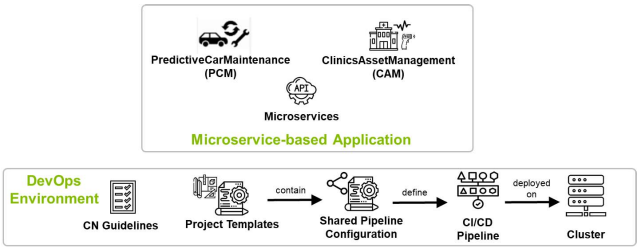

Figure 1. Topic Sketch

Figure 1 presents the two applications and the DevOps environment which we used to build, test, deploy and operate the applications. The environment includes a GitLab CI/CD pipeline. By providing reusable project templates that contain a CI/CD pipeline configuration, we can share most of our configurations by referencing the shared pipeline configuration. It consists of the framework-independent configuration files for the release and deployment stage as well as all the framework-specific configurations for several tech-

2642-6587/21/$31.00 ©2021 IEEE
DOI 10.1109/SOSE52839.2021.00020

nology stacks. This results in project templates with a single configuration which solely references the shared pipeline configuration. By implementing this structure, a single point of configuration can be achieved. This process reduces or even eliminates the increased management and maintenance effort required for a decentralized DevOps environment.

The present article is structured as follows: Chapter 2 presents the state-of-the-art microservices and DevOps concepts and technologies. Chapter 3 describes our DevOps environment, with a focus on the template concept that we introduced to simplify the configuration of the CI/CD pipeline by reuse of these templates. Chapter 4 provides a software developer's perspective on the DevOps environment by demonstrating the use of the templates through the example of a concrete microservice-based application. Subsequently, Chapter 5 discusses the actions that we have taken to protect our DevOps environment against attacks and introduces a pipeline-securing concept. Lastly, Chapter 6 summarizes the main results of our approach as well as the major research issues that we are currently working on.

2. Related Work

2.1. Microservices

Balalaie et al. [8] emphasizes the synergy of microservices and DevOps, which have displayed equal rates of growth among IT concepts over the past several years. The main reason is that microservices enable a continuous deployment, which is one of the major goals of DevOps. The term "microservice" originated in 2011, and a group of software architects chose it as the most appropriate name in 2012. Lewis was among the first researchers to use this term in a presentation [9]. Three years later, Lewis and Fowler [1] published the first comprehensive description on microservices and the microservice style. Based on their work, Newman [10] published the first well-accepted and often cited book on the topic. One strong motivation behind the microservice architectural style is the inherent disadvantage of monolithic systems with respect to change cycles and complex deployment since any changes to each small part of the application require a complete rebuilding and deployment of the entire monolith [2].

2.2. DevOps

Chen [5] mentions that while the microservice architecture reduces the complexity of each service, it also increases the number of services. This requires an additional effort in terms of interactions and contracts between the teams. To mitigate such problems, Wilsenach [11] states that the culture of DevOps should contain no silos between development and operations. Although there is no precise definition of the term "DevOps", certain concepts, such as automation, collaboration and shared responsibility among teams that are often associated with it. Lwakatare et al. [12] analyzed literature surrounding this topic and identify the five dimensions of DevOps: collaboration, automation, culture, monitoring and measurement. Erich et al. [13] reveal that different companies have varied views on the definition of DevOps, but they all share the opinion that the team, culture and automation are key elements of the concept. Furthermore, Senapathi [14] states that one reason for the adoption of DevOps is that it improves user experience due to the shorter response times to customer requests as well as higher team productivity resulting from automation and the reduction of communication barriers. This can be achieved through agile methods and CI/CD pipelines. Riungu-Kalliosaari et al. [15] analyze how different companies have adopted the DevOps concepts. Problems in communication and motivation for change are major aspects that mitigate the adoption process. Steven [16] claims that DevOps focuses on the culture and roles within the teams, and that it complements the processes of agile methods and the software development cycle of CI/CD.

2.3. Container Orchestration

In their paper, Kang et al. [17] note the ways in which containerization benefits the microservice and DevOps approach. The orchestration of the microservices plays a major role in DevOps as well as the requirements of modern software systems. Asif Khan [18] from Amazon lists scheduling, state management, and service discovery as a the main characteristics of container orchestration. This enables rolling updates of the software without any downtime or the need for a secondary system like, as is often required in the case of blue-green deployment.

Such requirements are typically configured by the operational team. Kubernetes is the widely used form of container orchestration [19] [20] which comes with its own complexity and tools. Paul Bakker [21] a software architect at Netflix, observes a significant advantage in standardizing the operational aspects. He also mentions that it was not sufficient for their use case and difficult to debug or monitor without additional tools and knowledge.

2.4. CI/CD Pipeline

A microservice architecture enables the efficient use of the CI/CD concept [5], for which Shahin et al. [7] list several challenges. Beside the lack of suitable infrastructure, the authors also mention common problems such as a lack of investment, which involves cost, skills, and time for integration and maintenance. Team awareness and communication is another major challenge, since CI/CD requires the collaboration of the developers and operational teams in the companies. Furthermore, continuous deployment demands additional awareness, since it can directly affect a production system.

To overcome the issues in the adoption of CI/CD pipelines, we introduce a template-based pipeline approach in Chapter 3. This approach allows all developer to deploy

the services with nearly zero effort and test them in a setting that is close to the production environment.

Since attacks on the DevOps environment and its CI/CD pipeline can cause great damage to an organization, adequate security measures must be taken. Ullah et al. [22] present five security tactics to protect main components such as the respository or CI server of the pipeline from malicious attacks. The effectiveness of the protection is then assessed through a comparison with an unsecured pipeline. Alternatively, Bass et al. [23] introduce a process to decompose the pipeline into a set of trusted and untrusted components, which are then isolated from each other. An interesting aspect of this approach is the analysis of the access rights and network connections of the pipeline components. In Chapter 5, we apply this generic process to the CI/CD pipeline of our concrete DevOps environment.

3. Template-based DevOps Environment

Figure 2 introduces our template-based pipeline approach, which aims to lower the entry barriers for developers to provide applications as well as reduce the maintenance efforts of the operational team. This is achieved through the use of templates and the generalization and reuse of the pipeline steps. Through the GitLab internal referencing mechanism, single files of individual steps can be injected and shared across multiple projects, thereby effectively addressing the major issue of the management of multiple projects in [5]. When using the pipeline templates, we must also ensure that no breaking changes occur due to the maintenance and expansion of the templates. To prevent this problem, we have introduced a workflow that ensures compatibility. Another method to achieve the above mentioned goals is using project templates; these provide developers with a quick start to deploy their software in a Kubernetes cluster. In addition, we have also created a service provisioning concept that can the develop of multiple features for a single service, without conflicts or additional costs to the environment. Our approach therefore addresses many of the existing challenges [7] and ensures that quality standards are met.

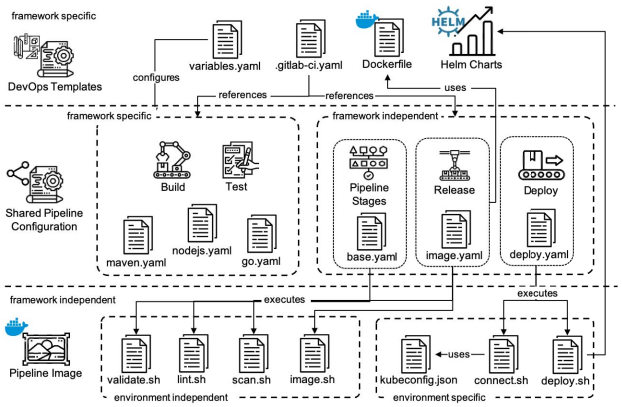

Figure 2. Template-based Pipeline Approach

3.1. Shared Pipeline Configuration

Each project template contains a predefined pipeline definition, which references the local project variables as well as the shared pipeline templates. A pipeline template provides all the necessary pipeline steps to execute a certain task. For example, 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 [24] allows the user to reference the relevant files—in our case the pipeline templates—in other projects. Additionally a reference to a tag or branch of the file can be set. The shared pipeline template mechanism make great use of this feature. During our research, we identified two types of pipeline steps: framework-independent and framework-specific. Framework-specific steps include source code compilation, test execution, code style checks, as well as security checks, which are further explained in Chapter 5. The two categories differ in the amount of customization and modification that they need to provide. Therefore, the developer has the option to extend the framework-specific pipeline templates using a local configuration in the project template.

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 has been introduced. For the workflow shown in Figure 3, each change to the pipeline templates or the deployment image triggers a different tests to prevent any negative influences. These tests include unit tests for the bash based script of the deployment image as well as the execution of a pipeline within a example project to test the interaction of all its pipeline steps. Afterwards, a code review is conducted to merge and release the changes into the production environment. To track each change, the semantic versioning specification [25] is used. This ensures that there are no breaking changes during a major version, and patches or features can be delivered to the developers without any modifications from their side. To allow the individual pipeline steps to be tested quickly and independently from GitLab, the logic has been encapsulated in the pipeline image. For example, image.sh and scan.sh contains all the logic for the build, push, and scan steps of the shared pipeline template for the Docker images. Therefore, each individual pipeline step can be tested locally, and any bugs can be fixed quickly using the image release mechanism provided by our shared pipeline template.

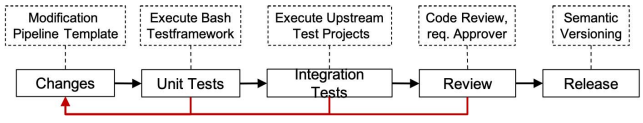

Figure 3. Maintenance Workflow for the Pipeline Templates

3.2. Project Templates

To support the developer in integrating and delivering the application, specific project templates adapted to the framework are offered. These templates contain all the artifacts required by the developer for the execution of a CI/CD process. Each project template includes a gitlab.yml file, which contains all the required templates from the shared pipeline for this framework. Since it is not possible to satisfy all the requirements for a project build with a single configuration, we provide a project-specific pipeline artifact. This allows us to encapsulate the project-specific build and tests the more static configurations of the pipeline templates. In addition, a framework-specific Dockerfile, which follows the best practices of Google [26], is provided. The Dockerfile builds and defines the port for the service, which is then referenced in the Helm chart of the project. The Helm chart in turn contains all the required Kubernetes manifests and dependencies as well as the values for the project. Predefined variables in the values.yaml file of the chart allow the user to add databases or other backing services to the project. To improve the usability, a recommended project structure is provided.

3.3. Application Provisioning

The provisioning for an application is predefined in the project template pipeline. Each template follows the process illustrated in Figure 4.

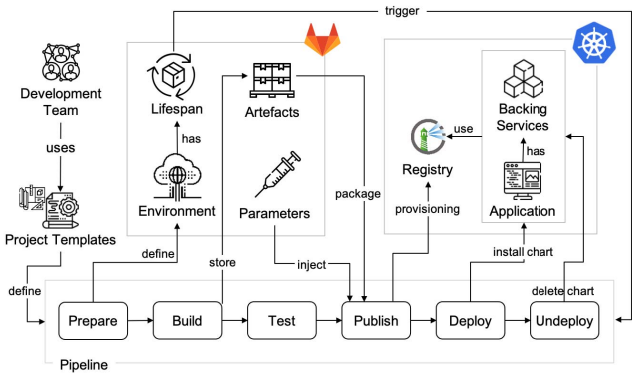

Figure 4. Application Provisioning Process from an Operational Perspective

In the first step, certain rules are checked, and a linting of the project is performed. Violations of company-internal guidelines and faulty configurations can be detected at the very beginning. In addition, 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. After the prepare step the software is built and tested. All build artifacts are passed on to the next steps. To enable a collision-free and configuration-free delivery, a large number of parameters are injected at the start of the publishing process. Therefore, predefined GitLab-specific and institute-internal environment variables are used. The developers have the authority to override these variables by setting them manually. Next, in the publish step, the Docker image is built, and 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. The flow of the pipeline execution varies depending on the application. Therefore, a distinction is made among version tags in GitLab as well as protected and feature branches, which differ in terms of the scope of the tests and the lifetime of the environment. This optimizes the build time and minimizes operating costs. In addition to these benefits, our approach allows allows the developer to focus more on their actual work and drastically reduces the effort spent on the operational aspects.

4. Developer's Perspective on the DevOps Environment

We developed the Predictive Car Maintenance (PCM) application using the project templates and shared pipeline configuration. The goal of this application is to predict maintenance events for cars. The design process resulted in three microservices: Vehicle, Diagnosis, and Predictive Maintenance. The application also has a frontend and a BackendForFrontend (BFF).

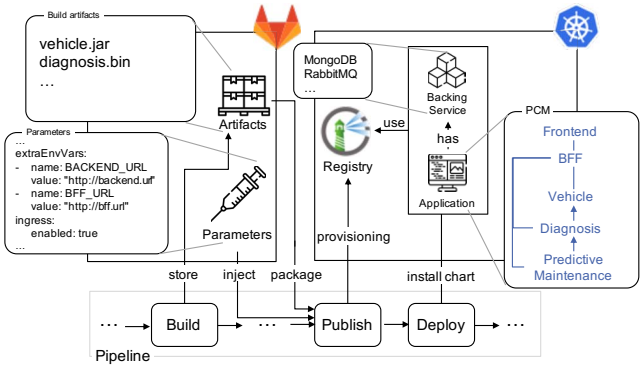

Figure 5. PCM Application in the DevOps Environment

The microservices are each maintained in a Git project. While the Vehicle microservice was developed using the Maven project template, the other microservices utilize the Node.js or Go templates. Figure 5 presents the PCM application in the DevOps environment. During the build step, the executable for each microservice is built and stored within GitLab (e.g., for the Vehicle microservice, it stores the vehicle-ms.jar file). The publish step takes the executables and parameters and creates the Docker image and Helm chart for each project before provisioning the image and the chart to the Harbor registry. For the Vehicle microservice, the parameters define a PostgreSQL dependency as well as the username and password for the database. Finally, the deploy step installs the Helm chart for each PCM service in the cluster. Backing services such as PostgreSQL and MongoDB are also installed in the cluster. Every Project Template allows the use of these backing services.

4.1. Why Project Templates Are Used

Coming from a monolithic architecture, microservice-based applications are a composition of multiple microservices (e.g., five microservices instead of one monolithic application). While this architecture positively influences flexibility, it also introduces complexity regarding DevOps [8]. However, in a monolithic application, a DevOps process has to be applied only to a single application. In a microservice architecture, this process is applied to all the microservices, thereby creating additional work for the developers. Moreover, within a team of developers, some must acquire knowledge on how the DevOps concepts work, how these concepts can be applied, and how projects that use these concepts have to be maintained.

The shared pipeline concept and the project templates help developers to simplify the DevOps process. Templates are available for several kind of projects; these include Go, Node, Java, Gradle, and Maven. Hence, development teams can use different programming languages for their projects. The templates assist developers with the specification of the DevOps environment without requiring them to know every detail of the underlying pipeline steps. For example, the developer does not have to know how the resulting Docker images are published to the registry or how the service is deployed to the Kubernetes cluster. However, the development team has to know how to configure, extend, and use the templates. To this end, the developer should be provided with extensive documentation.

4.2. How to Use Project Templates

Before a development team can start a new project, it has to decide on a programming language to use. Since the shared pipeline concept and project templates are currently developed using GitLab, a new Git project needs to be created, and the corresponding project template has to be imported. The Git project now contains a basic structure with the necessary files for a Java application. These include *variables.yaml*, *.gitlab-ci.yaml*, *Dockerfile*, and the *Helm charts* used to define the GitLab pipeline (see Figure 2).

Of course it is also possible to integrate the shared pipeline concept into an existing project. To do so, the existing project has to use an programming language that is supported by the project templates. The developer must add the abovementioned pipeline configuration files for the programming language to the existing project. For a Java application, the configuration files from the Maven template are used.

```
1  include:
2     - project: shared-pipeline
3       ref: "1"
4       file : base.yaml
5     - project: shared-pipeline
6       ref: "1"
7       file : image.yaml
8     - project: shared-pipeline
9       ref: "1"
10      file : deploy.yaml
11    - project: shared-pipeline
12      ref: "1"
13      file : maven.yaml
14    - local: 'variables.yaml'
```

Listing 1: Definition of the GitLab Pipeline Using the Shared Pipeline Configuration

By default, the pipeline of a project is defined in the *gitlab-ci.yaml* file, which contains references to the shared pipeline. An sample configuration file can be found in Listing 1. Lines 2, 5, 8, and 11 reference the configuration files defined by the shared pipeline, while lines 3, 6, 9, and 12 indicate the version to which the referenced configuration files have been set. A project may reference different versions of the shared pipeline. If version 1 is referenced, the master branch of the shared-pipeline project is used. Line 14 references the local *variables.yaml* file. To configure the project template, the developer has to define the parameters in *variables.yaml*. For the Vehicle microservice, the build variables *MVN_BUILD_CLI_OPTIONS* and *MVN_TEST_CLI_OPTIONS* can be set. These variables are used during the build and test stage, respectively. If a developer chooses not to set the variables, the default values are used, and the pipeline can be run without any additional configurations. If the developers require an optional Kubernetes Ingress (i.e., enable external access), this has to be enabled in the configuration file of the Helm chart.

If a developer requires additional steps for a single project, the GitLab pipeline can be extended by adding the steps to the project pipeline configuration. For example, the Vehicle microservice uses an additional step to trigger integration tests with the other PCM microservices. If the developer needs a template for a different programming language (such as C++) that is not currently available, they have to contact the pipeline operator to create a new project template and the corresponding framework-specific file defining the build and test stages for the shared pipeline configuration. This ensures that quality standards are maintained and that framework specific knowledge is centrally held by the pipeline operators.

When a developer pushes a commit to the repository, the pipeline is triggered, and they are presented with a detailed pipeline overview in GitLab, as depicted in Figure 6. The developer can see the execution status of the pipeline (i.e., whether it passed or failed) and the currently executed stage. A detailed log is available for each step. Hence, if there is a failure at any stage, the developer is notified and can review

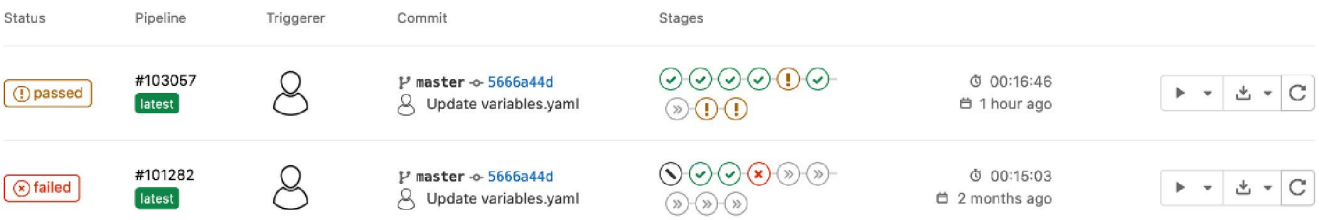

Figure 6. GitLab Pipeline Execution Overview

the logs to find the error. It is also possible to manually trigger the pipeline or re-run an exact copy of a previously executed pipeline so as to locate and fix bugs in the pipeline template.

4.3. Developer Experiences

The shared pipeline concept has been used successfully to build a variety of microservices. As a result, experiences with the shared pipeline were used to identify the existing problems. Overall, the templates have been positively received by the developers. Simple issues such as naming conventions in configuration files (e.g., paths or special characters) can lead to failures in the execution of the pipeline. Developers reported that the execution is quite complex and can be difficult to understand. Moreover, the documentation seems to be too distributed. These problems have been fixed, and each project template now has comprehensive documentation.

Some developers noted that the shared pipeline concept can introduce a single point of failure. Since all the projects currently reference the same shared pipeline configuration, a bug introduced into the central pipeline configuration consequently has an effect on every project. The pipeline might fail to run, and important updates might not be deployed as a result. Furthermore, a pipeline might run successfully but fail to deploy properly. This in turn can cause the development team to be unable to directly access the shared pipeline configuration or deploy projects manually to the cluster. While the aspect of the shared pipeline as a single point of failure is valid, it is unavoidable when designing the concept, and we believe that the overall advantages of the shared pipeline (such as central configuration) outweigh the drawbacks. Furthermore, developers can reference a version of the shared pipeline that remains unchanged over time (unless the latest version is referenced). This also allows them to quickly fix bugs for a specific pipeline version.

In general, management of access to the cluster, pipeline runner, or project can be considered a problem. Moreover, the developers complained about the time it takes to fully execute the pipeline. Although the execution time depends on the number of pipeline steps and the complexity of the project, this factor should be kept in mind when adding additional steps.

4.4. Current Work: Microservice Developer Portal

Developers of microservice-based applications may wonder if their microservice has been deployed correctly or which microservice are running at a given moment. In a Kubernetes system, these questions are not trivial, and cluster access is required to answer them, raising additional security concerns. Therefore, we are currently developing a *MicroserviceDeveloperPortal (MDP)* to provide developers with all the information that they need regarding their software environment. This portal provides users with a list of all the running microservices and details on each of them, including the address, API specification, service dependencies, personnel responsible, and health state of the service.

Manually created documentation tends to become outdated [27] and we do not want to create another system for developers to maintain. Instead, the project templates are used to create a project structure that allows them to automatically extract the relevant information, such as whether dependencies can be derived from the *values.yaml* file. The shared pipeline concept will add a registration step after a successful deployment to notify the MDP about the newly deployed microservice and deliver the extracted data. This ensures that the documentation will always be up to date with the code base from which the service is built.

5. Security Aspects of the DevOps Environment

Almost any application can be targeted by malicious attacks. In addition to the application, its deployment environment should also be protected against this threats [28]. Therefore, an advanced security concept should be part of a state-of-the-art DevOps environment. Continuous security testing can be integrated into the CI/CD pipeline [29]], and security checks such as dependency scanning should not require manual initiation on the developer's machine. Instead, this should be done in a reproducible and transparent manner within the CI/CD pipeline. Generally, when automated tests (such as unit tests) run in a CI/CD pipeline, their execution does not depend on the developer's local environment. Moreover, a significant amount of communication overhead is eliminated, as the pipeline execution status is clearly visible to all the team members.

These advantages are also relevant when running security checks within the CI/CD pipeline. Specifically, the

concept of *breaking the build* can be applied to security. Ideally, this concept improves security awareness among developers and encourages them to prioritize related issues. However, placing such a strong focus on application security leads to a fundamental question: What if the pipeline itself is not secure?

5.1. Security-First Pipeline Structure

The CI/CD pipeline can be considered the final element of a software supply chain [23]. Therefore, establishing a high level of trust towards the pipeline is essential. As trust in all pipeline components is infeasible in practice, DevOps teams are strongly discouraged from using monolithic pipelines in which components are not isolated from each other. This is because any untrusted component that is not sufficiently isolated may compromise the security of the entire pipeline. Accordingly, the pipeline introduced in Chapter 3 provides a high degree of isolation. 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. Such an architecture has significant security advantages. In a monolithic pipeline, each step operates on the same file system as all the other steps. Sensitive data can easily be leaked to an untrusted and subsequently executed pipeline step. For example, temporary files might be created and not properly cleared on completion. In a decomposed pipeline, this is not a problem because an explicit declaration is needed for each file or directory in order to make it available to subsequent steps, hence accidental leakage of data to untrusted components is much less likely to occur. An example of an explicit artifact declaration that is stored in a specific directory is shown in Listing 2. In this example, all the binaries generated by the step build are supposed to persist. Therefore, files in the `bin` directory are declared as artifacts, while all other files are automatically deleted when the container stops running. To make the configuration even more explicit, the directory name can be replaced with a specific file name. However, this would require additional effort when the build output is changed.

```
1 golangbuild:
2     stage: build
3     [...]
4     artifacts:
5         when: on_success
6         paths:
7         - bin
```

Listing 2: Explicit Declaration for File Persistence

In addition to the decomposed pipeline architecture, the template-based definitions explained in Chapter 3 also improve security. Even in an environment with many different projects, some security improvements can be implemented in the template files that are used by all the projects. For example, a dependency scan step can be introduced to all the 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.

5.2. Pipeline Security Issues in Practice

Although a decomposed pipeline structure provides strong security advantages, it falls far short of solving all pipeline-related security issues. Figure 7 presents an overview of security issues in the pipeline. The main issues that have been identified are *(1)* insufficient handling of credentials, *(2)* unrestricted outbound network access, and *(3)* insufficient isolation from production applications.

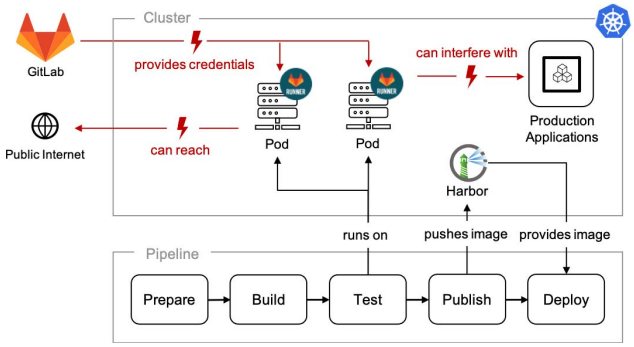

Figure 7. Pipeline and Related Security Issues

5.2.1. Credential Management. Most pipeline steps need access to some kind of external service. For instance, the step publish has to access a Harbor server in order to upload new Docker images. Harbor [30] is a Docker image repository that provides some additional features, such as a graphical user interface and support for vulnerability scanning. A repository access token is needed to access its API.

In the pipeline, credentials are provided through GitLab environment variables, which are made available to the pipeline containers. However, under the default settings, all such variables are available to all the pipeline steps. This can cause highly sensitive credentials to be visible to entirely unrelated steps; for instance, a simple source code formatting check may have access to the credentials used for deployment. This strongly violates the principle of least privilege [31]. Fortunately, GitLab provides environment labels [32] that can be assigned to the pipeline steps. Environment variables may then be restricted to a specific label. While this approach may be somewhat clumsy to implement due to the potentially tedious label assignment process, it does enable step-specific credentials.

5.2.2. Unrestricted Network Access. Some pipeline components require access to the public Internet. For example, downloading third-party dependencies is an important use case for this requirement. However, other components may not need to access any external service, apart from GitLab for the reporting of results. Again, the example of a source code formatting check may be used to illustrate this point.

By default, GitLab's Kubernetes pipeline executor does not impose any outbound networking restrictions on pipeline containers. If no default blocking policy is defined in the Kubernetes cluster, the pipeline containers can communicate freely with the outside world. Moreover, network isolation is not guaranteed between pipeline containers and production applications running on the same cluster. The potential effects of these issues include unwanted interference with production applications and leakage of data to external agents. Moreover, a sophisticated attacker may even attempt to access the Kubernetes cluster directly using a reverse shell if they manage to inject code into a pipeline component. Therefore, blocking all unnecessary outbound network access may be a requirement in highly sensitive environments.

To the best of our knowledge, step-specific networking restrictions are currently not possible in a standard GitLab- or Kubernetes-based pipeline. This is because Kubernetes network policies cannot differentiate between the containers responsible for different pipeline steps. In a true least-privilege approach, this limitation needs to be overcome. This can be done by slightly modifying the GitLab pipeline implementation to tag each container with the name of the step that it executes. Kubernetes container tags are essentially plain key-value pairs assigned to these containers. 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. Another limitation is that Kubernetes network policies are IP-based, and there is no direct method to control access to the services available via dynamic IP addresses. In this case, the responsible DevOps team may either use a proxy server or grant access to whole IP ranges. Nonetheless, the above solution provides a simple and effective way for DevOps engineers to set up fine-grained network access control in many cases. Since the default implementation makes a granular level of control impossible, this solution is already a significant improvement.

5.3. Pipeline Dependency Scanning

Once a robust and well-structured CI/CD pipeline has been established, it can be used as a basis for further security measures. This section focuses on a highly relevant use case: dependency scanning. The term refers to the automated auditing of the third-party dependencies of an application. The motivation for doing so is quite clear: modern dependency ecosystems such as Node Package Manager (NPM) contain hundreds of thousands of packages, creating a huge attack surface. In 2019, the average NPM package had about 87 cumulative dependencies [33]. Here, the *cumulative* means that the transitive closure of dependencies, including indirect dependencies, are considered. Critically, vulnerabilities in both direct and indirect dependencies can severely impact application security.

Due to the substantial scale of package ecosystems and the often large number of cumulative dependencies, it is difficult to manually audit all the dependencies. Ideally, third-party dependencies should only be added to an application if absolutely necessary. Enforcing this as a rule can prevent related security issues from occurring in the first place. However, existing projects may be deeply invested in using many different dependencies that cannot be easily removed. In such cases, dependency scanning can help engineers to keep a security overview. Various automated solutions such as Snyk [34] or Trivy [35] can be used for this purpose. These solutions all search vulnerability databases for matches between application artifacts and security advisories. The inner workings of dependency scanners are beyond the scope of this article, but it is important to note that they specifically target known vulnerabilities.

This paper does not intend to recommend any specific dependency scanner; rather, it aims to demonstrate *how* a scanner can be integrated into the pipeline. For practical reasons, dependency scanning should not be implemented separately for each project; the integration should be available to all projects by default. This is exactly where pipeline templates reveal their full potential. Specifically, an additional step vulnerability scan can be defined in the shared pipeline. By default, this step is executed in all the project-specific pipelines. Alternatively, an pipeline step opt-in can be subsequently defined. In that case, an environment variable must be set for each project in which the step is to be included. The opt-in approach may be useful to organizations in which a large number of existing projects have known security issues that cannot be fixed immediately.

The vulnerability scan step uses the Harbor dependency scanning functionality, which scans entire Docker images rather than plain source codes. This also implies that the scans are performed outside the pipeline. The pipeline itself is only responsible for initiating a scan, waiting for the scan results, and either continuing or aborting execution once the results are available. An overview of the architecture is given in Figure 8.

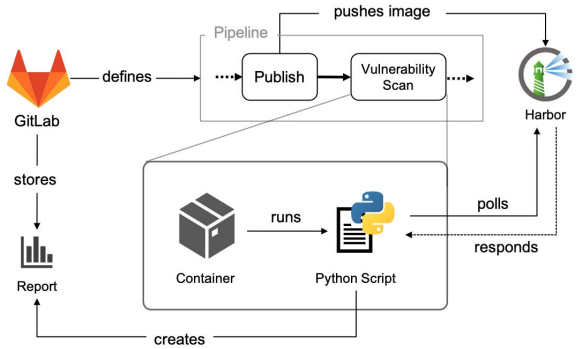

Figure 8. Pipeline Vulnerability Scanning

As Figure 8 illustrates, the vulnerability scan step is based on a Python script that interacts with the Harbor HTTP API. When it is run, the script initiates a scan of the

previously pushed image in Harbor. Subsequently, it busy-waits for a result by repeatedly polling the Harbor API until the result is available. While this solution may not be the most technically elegant, a more advanced approach based on webhooks would require more complex and error-prone configurations in Harbor and GitLab. The API call used for polling requires very little resources, as it only has to check if a result is available. Therefore, scalability is unlikely to become an issue in this approach.

Finally, 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.

One practical issue is that the Harbor API is stateless, i.e., the Harbor server does not have any information about the execution status of the pipeline. Therefore, the developer needs to be ensured that a pipeline execution will always receive the result of the same scan that it initiated. Otherwise, it might receive outdated scan results belonging to a previous execution of the pipeline. It is possible that while the content of a source code repository may have remained the same, the scanner used by Harbor or its vulnerability database may have changed. Therefore, merely tagging Docker images with commit hashes is not enough to uniquely identify the scan results. Instead, the step vulnerability scan relies on the GitLab CI_PIPELINE_IID environment variable, which is a project-level unique pipeline identifier. The value of this variable increases incrementally with each new pipeline execution. The Docker images are then tagged with this value, allowing the scan results to be uniquely traced back to a single execution of the pipeline.

Although the integration of the Harbor vulnerability scan into the CI/CD pipeline requires some effort, its advantages outweigh the implementation issues. The most important advantage is that Harbor ensures interoperability between pipeline and scanner implementation. As a result, a concrete scanner implementation can be replaced without changing the pipeline.

Developers may benefit even further from having Harbor as a middleware between the pipeline and the scanner. One crucial reason is that Harbor provides the option to ignore specific vulnerabilities identified through a Common Vulnerabilities and Exposures (CVE) identifier, which can be applied on a per-project basis. The option to ignore a vulnerability may appear counterproductive at first glance, but it can in fact hold pragmatic value for developers. As dependency scanners base their analysis solely on metadata-containing artifacts such as package lock files, the relevance of a vulnerability to a specific codebase usually cannot be determined automatically. As an example, a specific package may only be vulnerable if used in combination with untrusted (e.g., user- generated) input. Conversely, the input data of said package may always be considered safe in the scope of a particular project. In this case, the developers may choose not to immediately upgrade the package, especially when confronted with compatibility or time constraints. Obviously, ignoring a vulnerability issue may lead to a slippery slope and should not be done without thorough inspection. Clear guidelines for such cases should be defined with input from the developers. It is worth noting that the dependency scanning implementation proposed above is a rather invasive security measure for developers. Acceptance among developers is essential, and security absolutism may be an obstacle to such approval. The ultimate goal is to help developers deliver more secure software while minimizing any impedance to their workflow.

Finally, the architectural solution proposed in this paper may not be applicable to all environments. One caveat is that Harbor does not currently support the use of more than one scanner implementation at a time. This may constitute a significant limitation for some teams, due to platform requirements that cannot be fulfilled by a single scanner. In this scenario, the developers may decide to implement Harbor vulnerability scanning only for a subset of the projects or to integrate a scanner without using Harbor. Nonetheless, the Harbor-based approach can be useful for many teams, and it remains a valuable security achievement for both developers and DevOps engineers.

6. Conclusion and Further Work

The DevOps concept not only supports the integration of operational features but can also contributes to the development of sustainable software products. It allows users to automate and generalize aspects of the development process and operational concept. In terms of implementation, a thorough understanding of both areas is necessary to find a suitable measure between generalization and customization. Failing to do so can have negative effects on the acceptance of the approach by developers and operations.

6.1. Conclusion

Through the DevOps workflow introduced above, we provided an environment that supports the deployment process of microservices and enforces security patterns in their implementation. One requirement of the approach is the delivery of the microservice application in a container orchestration system such as Kubernetes. Since most developers are not familiar with Docker or Kubernetes, a 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. Publishing, delivery, as well as the base of the delivered artifacts were managed through the central pipeline templates. 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. We effectively mitigated the most common security issues for all CI/CD pipelines. Additionally, the integrated vulnerability checks created much higher awareness of security issues during development and

forced the developers to fix critical vulnerabilities as soon as possible. As a result, the number of attack vectors targeting the pipeline and the microservices could be significantly reduced, making them more secure.

6.2. Further Work

Aside from the positive aspects of the approach presented, developers missed the insight into their own running microservices and those of others. By not allowing them to access the service environment, attack vectors were drastically reduced, but at the cost of usability on the developer side. The permission restrictions meant that the services and interfaces of these microservices were invisible and had to be communicated between the individual teams. Since communication is essential for the reuse of microservices, we want to address this issue in further research. To overcome the problem, a DevOps-oriented mMicroservice Develop Portal (MDP) should be created to further support the work between the developer and operational teams. The MDP can showcase the mutual benefits of a DevOps-based approach and well-defined engineering processes.

References

[1] M. Fowler and J. Lewis, "Microservices," http://martinfowler.com/articles/microservices.html, Thoughtworks, Tech. Rep., 2015.

[2] S. Newman, *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*. O'Reilly Media, Inc., 2019.

[3] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[4] P. Giessler, T. Frauenstein, M. Gebhart, and S. Abeck, "API Design Guidelines for Resource-oriented Web APIs," in *International Journal of IEEE Transaction*, vol. 11, no. 2, 2015.

[5] L. Chen, "Microservices: Architecting for Continuous Delivery and DevOps," in *Proceedings - 2018 IEEE 15th International Conference on Software Architecture, ICSA 2018*, 2018, pp. 39–46.

[6] Y. Al-Dhuraibi, F. Paraiso, N. Djarallah, and P. Merle, "Elasticity in Cloud Computing: State of the Art and Research Challenges," in *IEEE Transactions on Services Computing*, vol. 11, no. 2, 2018, pp. 430–447.

[7] M. Shahin, M. Ali Babar, and L. Zhu, "Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices," *IEEE Access*, vol. PP, 03 2017.

[8] A. Balalaie, A. Heydarnoori, and P. Jamshidi, "Microservices Architecture Enables DevOps: an Experience Report on Migration to a Cloud-Native Architecture," *IEEE Software*, vol. 33, pp. 2–12, 2016.

[9] J. Lewis, "Micro services - The Java Way," http://2012.33degree.org/talk/show/67, Thouhtworks, Tech. Rep., 2012.

[10] S. Newman, *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media, Inc., 2015.

[11] R. Wilsenach, "DevOps Culture," https://martinfowler.com/bliki/DevOpsCulture.html, martinfowler.com, Tech. Rep., 2015.

[12] L. E. Lwakatare, P. Kuvaja, and M. Oivo, "An Exploratory Study of DevOps: Extending the Dimensions of DevOps with Practices," 08 2016.

[13] F. Erich, C. Amrit, and M. Daneva, "A Qualitative Study of DevOps Usage in Practice," *Journal of Software: Evolution and Process*, vol. 00, 06 2017.

[14] M. Senapathi, J. Buchan, and H. Osman, "DevOps Capabilities, Practices, and Challenges: Insights from a Case Study," 06 2018, pp. 57–67.

[15] L. Riungu-Kalliosaari, S. Mäkinen, L. E. Lwakatare, J. Tiihonen, and T. Männistö, "DevOps Adoption Benefits and Challenges in Practice: A Case Study," in *PROFES*, 2016.

[16] J. Steven, "What's the difference between agile, CI/CD, and DevOps?" https://www.synopsys.com/blogs/software-security/agile-cicd-devops-difference, Synopsys, Tech. Rep., 2018.

[17] H. Kang, M. Le, and S. Tao, "Container and Microservice Driven Design for Cloud Infrastructure DevOps," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, 2016, pp. 202–211.

[18] A. Khan, "Key Characteristics of a Container Orchestration Platform to Enable a Modern Application," *IEEE Cloud Computing*, vol. 4, no. 5, pp. 42–48, 2017.

[19] Sysdig, "2019 Container Usage Report," https://dig.sysdig.com/c/pf-2019-container-usage-report, Sysdig, Tech. Rep., 2019.

[20] Portworx and A. Security, "2019 Container Adoption Survey," https://portworx.com/wp-content/uploads/2019/05/2019-container-adoption-survey.pdf, Portworx and Aqua Security, Tech. Rep., 2019.

[21] N. Paul Bakker, Software Architect, "One year using Kubernetes in production: Lessons learned," https://techbeacon.com/devops/one-year-using-kubernetes-production-lessons-learned, TechBeacon, Tech. Rep., 2020.

[22] F. Ullah, A. J. Raft, M. Shahin, M. Zahedi, and M. A. Babar, "Security Support in Continuous Deployment Pipeline," in *International Conference on Evaluation of Novel Approaches to Software Engineering*, 2017.

[23] L. Bass, R. Holz, P. Rimba, A. B. Tran, and L. Zhu, "Securing a Deployment Pipeline," in *IEEE/ACM 3rd International Workshop on Release Engineering*, 2015, pp. 4–7, https://doi.org/10.1109/RELENG.2015.11.

[24] Gitlab, "GitLab CI/CD pipeline configuration reference," https://docs.gitlab.com/ee/ci/yaml, Gitlab, Tech. Rep., 2020.

[25] T. Preston-Werner, "Semantic Versioning 2.0.0," https://semver.org/lang/de/, accessed on 2020-11-26.

[26] C. Google, "Best Practices für die Containererstellung," https://cloud.google.com/solutions/best-practices-for-building-containers, Google, Tech. Rep., 2020.

[27] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: the state of the practice," *IEEE Software*, vol. 20, no. 6, pp. 35–39, 2003.

[28] V. Mohan and L. Othmane, "SecDevOps: Is It a Marketing Buzzword? - Mapping Research on Security in DevOps," in *2016 11th International Conference on Availability, Reliability and Security (ARES)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2016, pp. 542–547. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ARES.2016.92

[29] L. Bass, I. Weber, and L. Zhu, *DevOps: A Software Architect's Perspective*, 1st ed. Addison-Wesley Professional, 2015.

[30] H. Authors, "What is Harbor?" https://goharbor.io.

[31] R. S. Sandhu and P. Samarati, "Access control: principle and practice," *IEEE Communications Magazine*, vol. 32, no. 9, pp. 40–48, 1994.

[32] GitLab, "GitLab CI/CD pipeline configuration reference ¿ *environment*," https://docs.gitlab.com/ee/ci/yaml, accessed on 2020-10-14.

[33] R. Vaidya, L. De Carli, D. Davidson, and V. Rastogi, "Security Issues in Language-based Software Ecosystems," https://arxiv.org/pdf/1903.02613.pdf, 03 2019.

[34] Snyk, "Snyk Website," https://snyk.io.

[35] A. Security, "Trivy," https://github.com/aquasecurity/trivy.