University of Zurich[UZH]

INDEPENDENT STUDY – Communication Systems Group, Prof. Dr. Burkhard Stiller

Quantitative Maturity Assessment of DevSecOps Practices Using Metrics

Raphael Wäspi
St. Gallen, Switzerland
Student ID: 18-918-938

Supervisor: Jan von der Assen
Date of Submission: August 8, 2024

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

ifi

Independent Study
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: http://www.csg.uzh.ch/

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, _____

Signature of student

Abstract

The transition from a traditional Software as a Product model to a Software as a Service model has transformed the software development industry. It has enabled continuous improvement and deployment without the need for repeated deployment on the client side. However, despite these advances, it remains a challenge to integrate robust security practices into the DevOps space. Security is often seen as secondary in DevOps, which leads to potential vulnerabilities.

This study aims to fill this gap by automating the collection and analysis of DevSecOps metrics through the development of a prototype. The prototype was developed to fulfil the identified need for a flexible and configurable solution that can be customized to different DevSecOps environments. A comprehensive literature review was used to analyze key metrics and existing tools to influence the design of the tool. The prototype has a modular architecture that is already equipped with the first data collector and enables visualization by Grafana. This design enables the monitoring and evaluation of security practices and ensures that security is embedded throughout the development cycle.

Zusammenfassung

Der Übergang vom traditionellen Software as a Product Modell zum Software as a Service Modell hat die Softwareentwicklungsbranche grundlegend verändert. Das System ermöglicht eine kontinuierliche Optimierung und Bereitstellung, ohne dass auf Kundenseite wiederholte Implementierungen erforderlich sind. Trotz dieser Fortschritte bleibt es jedoch eine Herausforderung, robuste Sicherheitspraktiken in den DevOps-Bereich zu integrieren. Die Sicherheit wird bei DevOps oft als zweitrangig angesehen, was zu potenziellen Schwachstellen führen kann.

Diese Studie zielt darauf ab, diese Lücke zu schliessen, indem die Erfassung und Analyse von DevSecOps-Metriken durch die Entwicklung eines Prototyps automatisiert wird. Der Prototyp wurde entwickelt, um den Bedarf an einer flexiblen und konfigurierbaren Lösung zu erfüllen, die an verschiedene DevSecOps-Umgebungen angepasst werden kann. Im Rahmen einer umfassenden Literaturrecherche wurden Metriken und bestehende Tools analysiert, um die Grundlage für die Entwicklung des Prototypen zu schaffen. Der Prototyp hat eine modulare Architektur, die bereits mit dem ersten Collector ausgestattet ist und die Visualisierung durch Grafana ermöglicht. Dieses Design erlaubt die Überwachung und Bewertung von Sicherheitspraktiken und gewährleistet, dass die Sicherheit in den gesamten Entwicklungszyklus eingebettet ist.

Acknowledgments

First and foremost, I want to thank my supervisor Jan von der Assen for his continuous effort throughout this study. Thanks to his support and inputs in technical and academic matters, I was able to complete this Independent Study.

Furthermore, I would like to extend my gratitude to Prof. Dr. Burkhard Stiller who gave me the opportunity to do this Independent Study at the Communication Systems Group (CSG).

Contents

Chapter 1

Introduction

DevSecOps, short for Development, Security, and Operations, is an approach for software development that integrates security practices seamlessly into the DevOps workflow. It prioritizes security throughout the software development lifecycle, enabling teams to identify and mitigate risks proactively. DevSecOps fosters collaboration between development, security, and operation teams, leading to faster delivery of secure software. Research on metrics for assessing DevSecOps maturity is crucial for organizations to measure their progress, identify areas for improvement, and strengthen their security posture within the DevOps framework. This independent study aims to improve DevSecOps practices. Therefore, the following sections describe the motivation for this independent study and the scope and structure of this report.

1.1 Motivation

The motivation for this work comes from the realization that there is a significant gap in the tools available to automate the collection and analysis of DevSecOps metrics. While the security frameworks of security teams provide valuable guidance, the actual implementation and tracking of these frameworks is often inadequate. While there are many tools designed for specific DevSecOps activities, there is no known tool that collects data from these various tools, analyzes it, and shows which applications are compliant with security policies.

1.2 Description of Work

This study aims to fill this gap by examining existing tools that attempt to address this problem and by exploring the common challenges in DevSecOps. The goal is to develop a prototype that overcomes these challenges and addresses the key requirements by incorporating the most commonly cited metrics in the literature. Rather than relying on manual checks, which can lead to inefficiencies and inconsistencies, this research aims

to create a more comprehensive solution. By exploring the most popular DevSecOps metrics and the tools currently available, a prototype will be developed that addresses the shortcomings of existing solutions and provides a more robust approach to automating the collection and analysis of DevSecOps metrics.

1.3 Study Outline

The first chapter introduces the topic of DevSecOps and emphasizes its importance for the integration of security into the software development cycle. It contains the motivation for this study, the objectives and an overview of the work carried out. The second chapter deals with the basic concepts of DevSecOps. It covers the historical context, the goals and benefits of DevSecOps and explains the shift-left principle. It also introduces the DevSecOps framework with its activities and metrics and discusses the benefits of adopting this framework. Chapter 3 provides an overview of the existing literature and tools related to DevSecOps. It describes the methodology used for the literature review, analyzes key topics such as metrics, existing software and challenges in DevSecOps and concludes with a discussion of the insights gained from this analysis. The fourth chapter presents the high-level and detailed component architecture of the proposed prototype. It describes the Configuration, Collector, Analyzer, Visualizer, MySQL and Grafana components and outlines their roles and interactions within the system. Chapter 5 focuses on the implementation details of the prototype. It covers the technologies used, including Python, Docker, MySQL and Grafana, and provides an overview of the code structure. This chapter provides a technical foundation for understanding how the prototype was built. The sixth chapter outlines the methodology used to evaluate the prototype, describes how the evaluation was conducted, and discusses the results. The final chapter summarizes the project and provides an overview of the key findings and contributions. It includes a conclusion derived from the evaluation and suggestions for future work. This chapter highlights the importance of the study and its potential impact on the DevSecOps field.

Chapter 2

Background

This chapter introduces the theoretical concepts required to understand this study. The first section explains DevSecOps in general, including its goals and benefits. The second part deals with the specifics of a DevSecOps framework and defines key terms such as DevSecOps activities and metrics.

2.1 DevSecOps

The software industry has shifted from developing software as a product to providing it as Software as a Service (SaaS) [1]. This transition brought challenges due to lack of collaboration between development and operations teams. DevOps emerged to address these challenges by promoting collaboration and aligning priorities across teams. However, many DevOps programs overlooked security [2]. DevSecOps aims to extend DevOps by involving security experts from the start, fostering collaboration among developers, operators, and security professionals. As a new trend, understanding DevSecOps practices and experiences is crucial for effective integration of security into DevOps processes [1].

The two main benefits of DevSecOps are speed and security, which enables development teams to deliver secure code faster and more cost-effective [3]. DevSecOps is based on the principle that everyone is responsible for security and aims to integrate security measures as early as possible in the software development lifecycle (SDLC). This approach is in contrast to traditional methods, where security aspects are often added at the end.

The paper [4] shows that addressing security issues in the design phase is much more cost-effective, as the relative cost of fixing a bug is 100 times lower than in the maintenance phase. The DevSecOps 'Shift Left' strategy therefore not only increases security and speed, but also offers financial benefits. By including security in DevOps, companies can achieve a more efficient and secure development process.

2.2 DevSecOps Framework

A DevSecOps framework provides a set of baseline requirements for each phase of the development lifecycle. The goal is for the security team to define measurable and achievable requirements for the development teams. An example of such a framework is the Open Web Application Security Project (OWASP) DevSecOps Verification Standard, an open source framework that provides baseline requirements for every software project [5].

This study refers to **DevSecOps activities** as the specific requirements that a framework has at each stage of the software development lifecycle. In the context of the DevSecOps verification standard, these are called streams. For example, an activity in the code/build phase could be Static Application Security Testing (SAST).

Metrics are required to determine whether an application meets the specifications of a DevSecOps activity. These metrics define the criteria that an application must fulfill in order to pass the activity. In the case of SAST, for example, a metric could require that the application has no critical vulnerabilities and no more than five high vulnerabilities.

Maturity levels are often used in DevSecOps frameworks to indicate different levels of compliance. Metrics with different difficulty degrees can be defined, which are called levels, in order to identify which applications need to meet certain standards in each activity. The DevSecOps Verification Standard Framework [5] also contains such a maturity level system. To keep with the SAST example, a level one application could have no critical vulnerabilities and no more than five high vulnerabilities. In contrast, a level two application must have no critical or high vulnerabilities, and so and so forth.

Chapter 3

Related Work

This chapter aims to provide an overview of the current research in DevSecOps, particularly focusing on metrics and existing software in this field. The first part outlines the methodology applied for literature research and selection, which includes the categorization of papers into distinct groups based on their common themes. Subsequently, the identified groups are closely examined, aiming to uncover their distinct features and contributions to the study. Finally, a discussion is provided to summarize the findings and highlight key insights.

3.1 Methodology

In order to investigate the relationship between DevSecOps metrics and existing software solutions within this domain, a literature review was conducted. This search included a review of literature databases such as Google Scholar, IEEE Xplore, ACM Digital Library. A combination of keywords, including `DevSecOps`, `Metrics`, `SSDLC`, `Measurements`, and `Monitoring`, was employed to ensure a comprehensive search strategy. Both individual and combined use of these terms was utilized to maximize the number of relevant articles.

To broaden the scope of the review, the search was extended to include topics related to security issues in the realm of DevOps. This approach was undertaken to get more papers addressing this subject matter.

Articles were included in the review if they addressed challenging factors, metrics, or software solutions within the DevSecOps domain. The inclusion criteria were assessed based on the title, abstract, and the chapters of the articles. In cases where clarity was lacking, the full text of the article was consulted to make a determination. Additionally, articles older than 15 years were excluded from consideration to ensure relevance to current practices.

At the conclusion of this process, a total of 15 papers were carefully selected, comprising 12 papers and three industry resources. These selections were made based on their significant contributions to one of the three primary themes explored in this study: metrics,

challenges, or software within the realm of DevSecOps. Consequently, six papers were classified under the software theme, four under metrics theme, and five under challenges theme.

In the final stage of the literature review, the references of each of the 12 selected papers were examined. The three industry resources were excluded from the process, resulting in the identification of a total of 613 references. In order to subject the reference to further analysis, it was necessary for the title to contain either the term "DevSecOps" or "Security." Subsequently, out of the 613 references, 63 papers underwent further analysis. It is crucial to acknowledge that a considerable number of the references were duplicates or papers that had already been included in the initial selection. Consequently, the 613 references did not represent 613 distinct papers. It is also important to mention that 217 references originate from one single paper, which increases the total number. The 63 papers that met all the requirements were then categorized into one of the three main themes. From these 63 papers, 18 papers were finally considered relevant for these categorisations, resulting in a total number of 33 papers where 6 were classified under the software theme, 14 under the metrics theme, and 13 under the challenges theme.

3.2 Analysis of Key Themes in DevSecOps

This section provides a more comprehensive examination of the main themes and presents the findings of the relevant literature in this area. The three themes identified are metrics, existing software and challenges. Metrics covers the various measures used to evaluate DevSecOps activities. Existing software discusses the tools currently available to support these activities, while challenges highlight the hurdles and issues in implementing and optimising DevSecOps practices.

3.2.1 Metrics

In the field of metrics, it is evident that there is a significant focus on this topic in the literature, with six papers and eight industry resources specifically addressing metrics in DevSecOps. Furthermore, one paper which is classified under the challenges main topic also touches upon metrics. This brings the total to seven papers and eight industry resources emphasising the importance of metrics in DevSecOps practices.

However, because the term `metrics` is very broad, there are different interpretations of what should be measured in the DevSecOps context. Consequently, the literature review not only found software security metrics, which are discussed in more detail in the Chapter 4, but also general software metrics and business metrics.

For example, one paper [6] does not focus on software security, but on the security business in general. For example, this paper defines the metric `reduction in hours spent resolving security issues`, which is not directly related to software security, but aims to improve the security process through automation. This is not bad in principle, but it is not a metric that was targeted in this context. This also applies to another paper [7]

which focuses on business metrics by establishing metrics for costs. Examples of such metrics are the mean cost to patch, cost of incidents or the mean incident recovery cost.

However, not only business-related metrics were mentioned, but also software metrics that are not necessarily part of application security. An example of this is a paper [8] that defined metrics like number of executable lines of source code, number of lines containing source code or number of the maximum nesting level of control constructs like if and while.

In addition, the different types of metrics measurements were also presented in one paper [9]. These include explicit ordinal risk assessments and assessments of several vulnerability dimensions as well as metrics based on counts, frequencies and densities.

Although most resources in the literature highlight key metrics for measuring software security, such as mean time to fix a vulnerability, number of security-related tickets, and the number of vulnerabilities in an application, it is important to acknowledge the existence of additional metrics within the field. While these metrics are potentially relevant to DevSecOps, they were not specifically examined in this independent study.

3.2.2 Existing Software

In terms of existing software, the literature review revealed five papers and one industry review relating to existing software in the DevSecOps field. This subsection introduces each tool in the selected publications and explains its core functionality. The aim of this is to provide a comprehensive overview of the existing software tools in DevSecOps.

The paper by [10] stated that software engineering needs a metric-driven approach to increase the quality, adaptability and security of a software and to decrease the time-to-market. It also proposes an architecture of such a metric-driven approach like in the ITEA3's MEASURE project [11] and H2020's MUSA Project where a centralized platform is dedicated for measuring, analyzing and visualizing metrics to get information about the software engineering process. This is done with measures using the Structured Metrics Metamodel (SMM) standard which are automatically collected by the SMM Engine. The metrics that can be collected are listed on their GitHub wiki [12]. While many metrics are predefined and fixed, they first define the metric and then specify how it can be collected. With this approach, you lose variability in the metrics and must use them as they are defined. After collecting the measures, it analyzes them and then provides a graphical representation of the information with reports defined by several dashboards. This is done because it can help managers in their decision-making process. However, it is important to note that the last update of this tool was in 2019, and most of the repositories are archived on GitHub, which leads to the assumption that this project is no longer used.

In another paper [13], a literature review found that while DevSecOps metrics such as deployment rates or lead times provide insight into the progress of software development, they are not sufficient to replace program control metrics for assessing progress, which would be possible with interactive visualization dashboards. For this reason the paper introduces a hypothetical Proof of Concept (PoC) where in a first step they collected data

automatically from Continuous Integration/Continuous Delivery (CI/CD) tools to track the planned, actual and projected completion. After that the results were displayed which should help to explore information gaps, need and data requirements for decision-making.

The paper by [14] proposes a self-service infrastructure for cybersecurity monitoring that is key to a DevSecOps culture. The paper refers to this as `monitoring as code`, as the development and operations teams are able to configure their own monitoring and alerting services based on security criteria, which should allow the silos between development, operations and security teams to be broken by opening up access to key security metrics. The monitoring is based on the detection of threats and anomalies from the application logs.

The paper [15] brought the state-of-the-art in SDLC and Secure Software Development Life Cycle (SSDLC). As part of their work, they defined a tool to manage SSDLC because they argue that in today's world, many decisions are made by different teams when a product is brought to market. Therefore, the managing SSDLC tool brings in metrics to measure the effectiveness of security controls and general security controls. This means that security activities are integrated into the application development process, such as code review, penetration testing and others. In addition, the paper points out that the business criticality of a developed application plays an important role in not unnecessarily influencing the development of an application that should not have such strict security controls.

In contrast to the tools mentioned above, the paper by [16] and the online resource [17] show technologies that should play a role in DevSecOps. Previously, there were tools that aimed to manage and monitor DevSecOps metrics, but these publications focus more on how to measure an application and obtain the data to create metrics. For example, the paper [16] defines that implementing and automating security scanning tools such as Snyk or StackHawk is an effective approach to secure applications. They suggest including tools for SAST and dynamic application security testing (DAST) in the CI/CD pipeline of all applications. The online resource [17] is similar but offers a broader collection of tools for each phase of the development lifecycle. For example, for the planning phase, he recommends a threat modelling activity where you can use tools such as the Microsoft Threat Modelling Tool or OWASP Threat Dragon.

3.2.3 Challenges

In total there were eleven papers and two industry resources that were classified under the main theme of challenges in DevSecOps. This subsection lists some of the challenges that were repeatedly mentioned in the publications.

The first challenge in the area of DevSecOps is culture, which is highlighted by several sources. First and foremost, [18] highlights the importance of creating a security-focused culture and the negative effects of a lack of management prioritisation and defined responsibilities. It also defines the problem that developers lose their autonomy when security authorities try to introduce security processes for the development, and that this can lead to conflicts between developers and security authorities. In addition, [19] highlights the

symbiotic relationship between cultural resistance and organizational structure, noting that the nomination of security champions can be a driver of cultural change as well as organizational structure change. So, organizational barriers in general is also a challenge, as discussed in [1] and [20], because it can hinder effective collaboration between security teams and the rest of the organization, which results in operational inefficiencies. However, [21] suggests that while unrestricted collaboration may pose risks, supervised collaboration with the security team can improve system security. In addition, [22] and [23] highlight the challenge of integrating individuals into cohesive teams that include developers, operations personnel, and security professionals. Finally, [3] defined the challenge that it is often not clear who is responsible for security in an organization and that it is important in DevSecOps to communicate the responsibilities for process security and product responsibility. Only in this way can developers and engineers become process owners and take responsibility for their work.

A key challenge in DevSecOps is the identification and implementation of appropriate security metrics. Despite that there might be various security metrics already available, as mentioned in [18], their integration in a DevSecOps team is still limited, possibly due to the lack of connection between developers and security engineers. Furthermore, [24] points out the danger of using inappropriate performance metrics for security assessment and stresses the importance of aligning metrics with the specific security goals of DevSecOps practices.

The availability and suitability of tools is also a major challenge. One aspect is the lack of solutions for visualizing security data, which leads to manual inspection at different stages [25]. In addition, according to [26], it is very important to have a consensus on the tool selection within the development teams. [27] and [18] describes that the use of security tools and securing all components with DevSecOps metrics such as secure coding, counting the number of security issues and various controls is considered a best practice, but very difficult to implement. Overall, promoting frequent security checks and using security dashboards, as suggested in [23], can help to track threats and dependencies effectively. In addition, the lack of automated testing tools, as highlighted in [28], [20] and [24], is a barrier to efficient security testing within the DevOps pipeline. Here, [20] states that defining security requirements and metrics such as Mean Time to Resolve (MTTR) or (Mean Time to Detect (MTTD) can help mitigate the organizational barrier in the area of automated testing.

3.3 Discussion

There is a lot of research on the theoretical foundations of DevSecOps, explaining what metrics to measure and how to secure applications within organisations. However, when it comes to practical implementation, specifically tools or PoCs that automatically collect and visualize these metrics, the research landscape becomes sparse. While there are several successful tools in the DevSecOps area, they often specialize in one aspect, such as automated security testing. This specialization can lead to a distribution of tools across an organisation, resulting in a loss of visibility into the security maturity of each application. As a result, there is a lack of solutions that bring together metrics from different tools in

one central location to give developers and security engineers a comprehensive overview of the current situation of all DevSecOps activities in all applications within an organisation. Despite this challenge, some papers have recognized the problem and attempted to develop software and PoCs to address it. These tools all have very good approaches and do some things very well, but there is no tool that combines these approaches. For this reason, the following chapters are about introducing a PoC that does just that.

This PoC is very similar in architecture and principle to the tool from [10]. It should automatically collect, analyze and then visualize metrics that a security engineer defines. However, unlike in [13], in this PoC the data should initially be collected centrally by a security team member and not separately in each CI/CD pipeline by the development teams. The advantage of this is that the developers only have to say where the data can be found. A trained security officer is then responsible for collecting the data, which should be fully automated. The advantage of this is that it is not necessary for several developers to become familiar with the tool, but only one security officer. This should prevent the conflict described in [18], since the work does not lie with the development team. Furthermore, as described in the managing SSDLC tool [15], the PoC should also be able to categorise applications according to their business criticality, so that not every application has the same security requirements. It should also be able to represent multiple security activities for each software development phase, as defined in [17]. These activities may be different for each organisation. In other words, it should not dictate activities, but be open to the types of activities that need to be tracked.

The fact that in this PoC it is completely arbitrary what you want to measure means that it is possible to measure organisational structure metrics, such as security champions. This is defined as a challenge in [19]. The fact that the metrics are presented visually, addresses the challenge described in [18] of a lack of management prioritization. Currently, it is often difficult to illustrate for application teams what they are missing and why this poses a security risk. By only being able to show it graphically, you can also communicate faster and better with the management. For example, the lack of automated testing tools mentioned in [28], [20], [24] should be pointed out as a problem. In addition, this tool also promotes collaboration between security and DevOps, but not to the level of full supervision in code and so on, which is also highlighted as a problem in [1]. However, the problem of the lack of tools cannot be solved with this PoC. This is because the PoC is a tool, as suggested in [23], that collects and analyzes data from other tools, but does not generate data for a specific DevSecOps activity as a tool. Nevertheless, it can serve as a guide to stimulate ideas and provide an overview of tools in DevSecOps.

This PoC is therefore a metrics-driven approach to improving software development processes by increasing software quality, adaptability and security, and reducing cost and time to market [10].

Chapter 4

Architecture

This chapter describes the architecture of the prototype developed in this independent study. The first section provides an overview of the high-level structure of the prototype and gives an insight into how the system works as a whole and how the various components interact with each other. This overview is intended to illustrate the overall concept and the data flow within the system. The second section deals with a more detailed examination of the individual components. Here, the specific functionalities, interactions, and implementations of the core components of the prototype are analyzed. It also shows which metrics can be collected in this prototype.

4.1 High-Level Architecture

As shown in Figure 4.1, the prototype consists of three core modules. The first and most important module is responsible for collecting, analyzing, and serializing the data, which is then sent to the second module, the database. The final module is then responsible for visualizing the data stored in the database. The design of these modules illustrates the intended separation with Docker containers and the interaction between them.

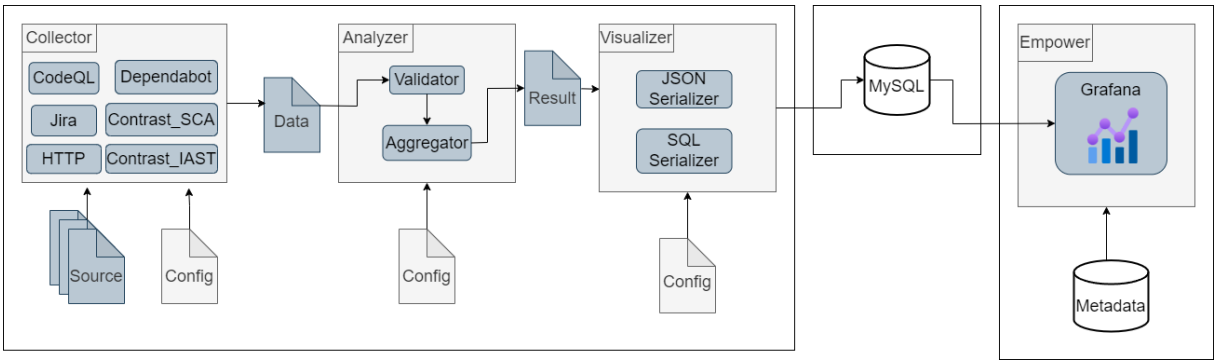

Figure 4.1: High-level Architecture

The entire prototype is called CAVE, which stands for *collector*, *analyzer*, *visualizer* and *empower*. Figure 4.1 illustrates the concept behind these components. The *collector* retrieves data from a specific location and processes it in a standardized format. This data is then sent to the *analyzer* which checks whether the data meets certain criteria. The *analyzer's* results are forwarded to the Visualizer, which processes the data for storage in a SQL database or in JavaScript Object Notation (JSON) format. In the final *Empower* step, the data stored in the SQL database is visualized to clearly show which metrics are met by each item.

It is important to note that the entire architecture is highly configuration-orientated. A robust configuration is crucial for enabling the functionality. Therefore, the *Collector*, the *analyzer*, and the *Visualizer* are configured using JSON files. These configuration files define which and how data is collected, how the *analyzer* checks the data against certain conditions, and where the *Visualizer* stores the results. This configurability makes the prototype highly customizable, which allows users to define their own metrics and criteria. This approach addresses some of the challenges discussed in Chapter 3.

4.2 Detailed Component Architecture

In this section, the individual components of the prototype are discussed in detail. This will illustrate their individual contribution to the overall architecture and how they work together to achieve the system's goals. In each subsection, the implementation and the parameters handled by the components are described in detail to ensure a complete understanding of their functionality and integration.

4.2.1 Configuration

Listing 4.1: Configuration file in JSON Example

```
{
"items": {
    "APP-1": {
        "sca-L1": {
            "collector": {
                "type": "dependabot_sca",
                "base": "https://api.github.com/repos/sumsumcity/webgoatOutdated",
                "params": {
                    "per_page": 5000,
                    "state": "open",
                    "severity": "critical,high"
                },
                "verify": false,
                "headers": {
                    "Authorization": "Bearer ${GITHUB_PAT}"
}
            }
        }
    }
},
"metrics": {
    "sca-L1": {
        "level": "1",
        "validator": [
```

```
            {
                "result": "fail",
                "prio": 1,
                "description": "Vulnerable library with severity critical or high is used"
            },
        ],
        "aggregator": {
            "missing": {"result": "pass", "description": "There are no dependabots alerts"}
        }
    },
    "sast-L1": {
        "level": "1",
        "validator": [
            {
                "result": "fail",
                "prio": 2,
                "description": "Critical vulnerability is present",
                "str:response.state": "open",
                "str:response.rule.security_severity_level": "critical|high"
            },
            {
                "result": "pass",
                "prio": 3,
                "description": "No critical vulnerability"
            }
        ],
        "aggregator": {
            "missing": {"result": "pass", "description": "CodeQL found no vulnerability"}
        }
    }
},
"serializer": {
    "type": "mysql",
    "host": "127.0.0.1",
    "database": "caveDB",
    "port": 3306,
    "user": "root",
    "pass": "${MYSQL_PASSWORD}"
}
}
```

As described in Section 4.1, the configuration is the centerpiece of this prototype, as it enables variability and facilitates extensions. This is the reason that all components must be configured. The configuration can be divided into three parts: items, metrics, and serializer. An example of such a configuration is shown in Listing 4.1.

The first part, **items**, defines all applications. It is called items because it can include not only applications but also other units, such as development teams. This system allows flexible definitions. According to Figure 4.1, this part configures the *collector* for each metric for each item. Listing 4.1 shows that a *collector* is defined for the application metric *sca-L1*.

The second part, **metrics**, implements the *analyzer*, which consists of the *validator* and *aggregator* components. It is important to note that all metrics are used for each relevant item. Any change to the metrics applies to all items, so metrics can easily be added without changing the configuration of the individual items. In Listing 4.1, the metric *sca-L1* fails if the *collector* returns a list of critical and high-risk vulnerabilities. An empty list results in success. For *sast-L1*, the *collector* returns all vulnerabilities and the *validator* determines the pass/fail status using tags such as `str`, `date`, `lt` (lesser than), `gt` (greater than). The priority key in the *validator* ensures that the *aggregator* takes the result with the lowest priority into account. For example, if there is an open critical or high-risk

vulnerability, the metric will fail. The *aggregator* can also use a `count` key to specify the number of vulnerabilities required for a failed result, which is useful for setting thresholds. In addition this configuration enables also the definition of DevSecOps activities [14] [17] with multiple maturity levels which was a key requirement in the paper [15] and [29]. Each activity can be performed multiple times, with different levels and requirements for the *validators*/*aggregators*. The flexibility of this PoC enables the measurement of organizational structure metrics, such as security officers, which is a challenge described in [30].

In the third and final part of the configuration, the *serializer* must be configured. In the current version of the PoC, users can choose to store the results either in a MySQL database or in JSON format. This flexibility allows users to select the storage format that best fits their needs and infrastructure.

4.2.2 Collector

Metric	Description	References
MTTR	Time to resolve a vulnerability	[31] [32] [30] [33] [34] [7] [29] [35] [36]
Mean Lead-Time (MLT)	Time between code commit and production	[37] [32] [30] [33] [36] [29] [35]
Number of Security Vulnerabilities	How many vulnerabilities does the app have	[37] [31] [33] [34] [7] [9]
Number of Deployments	How often code changes are deployed to production	[31] [32] [30] [33] [36] [29]
Number of security-related Tickets	Number of tickets with security label	[32] [30] [6] [36] [29]
Number of Failed Deployments	How many vulnerabilites does the app have	[32] [30] [36] [29]
MTTD	Time to detect a vulnerability	[31] [30] [36] [35]
Test Coverage	Percentage of test coverage	[31] [30] [33] [29]
Defect Burn Rate	How often vulnerabilities are deployed in production	[37] [38]

Table 4.1: Key Metrics for Evaluating DevSecOps Practices

The *collector* component is responsible for retrieving data from third parties and processing it in a standardized format. Each DevSecOps activity should, therefore, have a clear metric that is analyzed against the data automatically collected by the *collector*. Given the limited scope of this independent study, it was decided that the *collector* should focus on retrieving data for only a subset of the most important metrics. To identify these key

metrics, a comprehensive literature review was conducted. This review not only focused on metrics but also covered other relevant aspects as described in Chapter 3. The results of this literature review considering 33 papers, including the key metrics identified, are summarized in Table 4.1. However, when looking at the results, it should be noted that two papers are cross-referenced, meaning they have very similar metrics as they have taken the information from the other paper. This is due to the literature search described in Chapter 3.

When analyzing the nine metrics, it is noticeable that they can be divided into two categories. The first group measures the time between two events and calculates the average of these times as a metric. This group includes `MTTR`, `MLT` and `MTTD`. As described in the article [32], these metrics evaluate the stability of an application. The second group includes all other metrics listed in Table 4.1 that measure a specific variable. These metrics do not provide information about stability but rather about the current exposure of an application. This second group of metrics was, therefore, of greater interest for the first prototype. They allow for faster adjustments as they are binary (true or false) and do not require calculating average times. In addition, they can be obtained directly from third-party data without the need for calculating historical data. For this reason, only six metrics remained for the prototype.

However, it is important to note that almost any DevSecOps activity can have both stability metrics and current exposure metrics. For example, in patch management, you can measure the number of vulnerable libraries and the average time to fix these vulnerabilities, which must meet certain thresholds. However, to finalize the selection of metrics, it is important to consider meaningful DevSecOps activities and determine how these activities should be measured. The OWASP DevSecOps Verification Framework [5] provides valuable insights into possible activities. It should be emphasized that multiple activities focus on security vulnerabilities, with the metric `number of security vulnerabilities` playing a crucial role. In addition, the metric `number of security-related tickets` is relevant for the activity `Security Issues Tracking Design`. Therefore, these two metrics were selected for the first prototype.

Activity	**Metric**	**Collector**
REQ-004 Security Issues Tracking Design	number of security-related tickets	Jira
CODE-004 Static Application Security Testing (SAST)	number of security vulnerabilities	codeql_sast
CODE-005 Software Composition Analysis (SCA)	number of security vulnerabilities	contrast_sca dependabot_sca
TEST-003 Interactive Application Security Testing (IAST)	number of security vulnerabilities	contrast_iast
DES-002 Threat Modelling	Date, Content, etc	HTTP

Table 4.2: Interconnectivity between DevSecOps activities and the CAVE collectors

As in Table 4.2, the prototype contains several *collectors* that collect data from various technologies for DevSecOps activities and their metrics. The prototype uses Jira for issue tracking. It supports Contrast Security and CodeQL for all application security tests (SAST, IAST). Contrast Security and Dependabot are also used for Software Composition Analysis (SCA). It is important to mention that CodeQL and Dependabot were chosen because they are easy to activate on GitHub and, therefore, accessible for many applications. However, for threat modelling, an Hypertext Transfer Protocol (HTTP) *collector* was set up for which there are no reported metrics in this literature review. This *collector* was initially created to test whether it is possible to retrieve data with the planned architecture and analyze it accordingly. The test was successful and has shown that this is both possible and very useful, as it can collect metrics such as data or other information from any website. In the case of threat modeling, this HTTP *collector* can measure when the threat model was created to determine whether it is still up to date.

4.2.3 Analyzer

The *analyzer* is a key component of the prototype that is responsible for analyzing and interpreting the collected data in order to gain meaningful results. It consists of two main elements: the *validator* and the *aggregator*. These elements work together to ensure that the metrics collected from various sources are accurately evaluated and aggregated, which enables an analysis of DevSecOps activities.

The *validator* is used to evaluate the data collected by the *collectors* using various predefined criteria. It uses a number of tags such as `str` (for string comparisons with regex), `date` (for ISO date formats), `lt` (less than), and `gt` (greater than) to analyze the data. These tags enable flexible and precise validation rules that can be adapted to different metrics. For example, a metric may fail if a list of critical or high-risk vulnerabilities is returned, while an empty list would return a positive result. Each *validator* also contains a priority key (prio) that determines the importance of the validation result. This prioritization system ensures that the most critical validation results are considered first during aggregation.

The *aggregator* processes the validation results of the *validator*. It considers all validation results and uses the one with the lowest priority to determine the overall result. By combining the *validator* and *aggregator*, the *analyzer* provides a robust mechanism for measuring the security and compliance of DevSecOps activities. It enables organizations to define and enforce validation rules for all phases of the software development lifecycle to ensure that critical issues are detected and fixed immediately. This flexibility makes the *analyzer* an essential part of the prototype that supports the continuous improvement and maturation of security.

4.2.4 Visualizer

The *visualizer* is a key component of the prototype that transforms the collected and analyzed data into visual metrics. This visual representation addresses the challenge

described in [26], which highlights the difficulty of interpreting complex data sets without a clear and intuitive interface.

The *visualizer* supports various *serializers* to meet different data storage preferences. The *JSON-serializer* outputs the data in JSON format and provides a flexible and easily accessible way to view and share metrics. JSON serialization is ideal for integration with other tools and systems that support or use JSON data. In contrast, the *SQL-serializer* stores the data in a MySQL database and enables robust query and analysis capabilities. SQL serialization is suitable for environments that require structured data storage and complex query operations.

By visually displaying metrics, the *visualizer* improves the user-friendliness and accessibility of the prototype. This allows stakeholders to quickly understand the data and act accordingly. This approach not only makes the metrics easier to understand but also facilitates better decision-making and continuous improvement in the DevSecOps area.

4.2.5 MySQL

In the architecture, MySQL is provided in a separate Docker container, which enables isolated and robust data storage. This structure ensures the integrity and scalability of the data management system and enables efficient handling of large data sets. Within the database, the metrics and items must be inserted manually. This manual insertion process provides flexibility and allows users to add custom metrics and elements as needed. Each metric must have a `metricid`, and each item must have an `itemid` that matches the configuration. This is critical as these IDs serve as foreign keys in the results table and link the collected data to specific metrics and items.

4.2.6 Grafana

Grafana is integrated into the architecture to provide advanced data visualization functions. When using the *SQL-serializer*, Grafana can directly access the MySQL database. This allows Grafana to execute SQL statements to retrieve data. With this setup, users can create various dashboards and visual representations of the data that provide insights into metrics and performance. Grafana's flexibility in creating dashboards and data visualization makes it an invaluable tool for monitoring and analyzing DevSecOps activities, ensuring that key metrics are easily accessible and understandable for all stakeholders.

Chapter 5

Implementation

This chapter explains the technologies used to create the prototype described in Chapter 4. It describes not only the technologies used but also the structure of the source code.

5.1 Technology

This section provides an overview of the implementation process. It details the technologies that were used to ensure a clear understanding of the dependencies of the prototype. To focus on specific aspects of the implementation, the section is divided into several subsections.

5.1.1 Python

Python is an interpreted high-level programming language known for its clear syntax and good readability [39]. It supports modules, classes, exceptions, and dynamic data types, which simplifies code reuse and modularity. Python is particularly popular as an easy-to-use scripting language for rapid development, which makes it very attractive for various applications. For these reasons, Python was chosen for the development of the prototype in this independent study.

However, it is important to highlight that the prototype also relies on important dependencies and libraries, which are listed in the `requirements.txt` file on GitHub [40]. One essential dependency is the **requests** library, which simplifies the sending of HTTP requests [41]. Since the prototype contains a *collector* that retrieves data from various sources, the requests library is fundamental to its functionality. Another important library is **python-dotenv**, which reads key-value pairs from the `.env` file and sets them as environment variables [42]. The `.env.example` file shows the variables that the prototype uses for authentication with different data sources. In addition, the Python library **mysqlclient** enables the connection to the MySQL database and, therefore, the manipulation of tables and data [43]. Lastly, the **isodate** library is essential as it implements the analysis

of date, time, and duration according to the ISO 8601 standard, which is crucial for the validation component of the prototype [44].

5.1.2 Docker

Docker allows separating applications from infrastructure, which enables faster software deployment [45]. By allowing to package and run an application in an isolated environment called a container, Docker ensures isolation and security and allows multiple containers to run simultaneously on a single host. These containers are lightweight and contain everything needed to run the application, which removes the dependency on the software installed on the host.

As shown in Figure 4.1, each block represents a separate Docker container. Therefore, the prototype contains a `docker-compose.yaml` file in which three separate containers and two volumes are defined [40]. The main component of the CAVE tool is not included in a Docker container, as it is still under continuous development and it is therefore too early for a continuously maintained container. Instead, the MySQL database, the Adminer tool for managing the database, and Grafana operate in their own containers. MySQL and Grafana also use volumes to ensure data persistence even if the containers are restarted. In future updates, the CAVE tool, along with its *collector*, *analyzer*, and *visualizer* components will also be made accessible within a Docker container. This extension will enable full use of the tool in a Kubernetes environment.

5.1.3 MySQL

MySQL is the most popular open-source SQL database management system developed, distributed, and supported by Oracle Corporation [46]. It is a relational database that provides a robust and reliable solution for managing and organizing data. As an open-source database, MySQL is freely available and widely used in various applications and industries.

The widespread support of MySQL by various applications is a key factor why the CAVE tool also supports it, as it integrates seamlessly with Grafana without the need for additional plugins. The current database setup includes three tables: metrics, items, and results. The result table is populated by the *serializer* component and contains foreign keys that are linked to the metrics and items tables. It also contains a status field that shows the result of the *analyzer* (passed, failed, error, missing) along with a description and date. The metrics and items tables are not populated by the CAVE tool itself, as this task should be performed manually or by a separate script.

5.1.4 Grafana

Grafana is an open-source software that allows you to query, visualize, set alarms, and explore your metrics, logs, and traces wherever they are stored [47]. It provides tools to

transform your database data into insightful charts and visualizations. In addition, the Grafana plugin framework allows you to connect to various data sources, such as NoSQL and SQL databases, ticketing tools, and continuous integration and deployment tools like GitLab.

In the CAVE tool, we use Grafana with MySQL as the data source, which allows for automated querying and visualization of the database. Since MySQL is a built-in core data source in Grafana, the setup is ideal for environments where downloading additional plugins is not allowed [48]. Figure 5.1 shows a sample query in Grafana that specifies the data to be retrieved from the database. Grafana also provides built-in transformation features that allow users to customize the visualization of the data, for example by grouping it into matrices or concatenating fields.

Listing 5.1: Example Query in Grafana

```
SELECT r.*, concat(r.status, ": ", replace(r.description, '\n', ' ')) AS d
FROM caveDB.results as r
JOIN items AS i ON i.itemid = r.itemid
JOIN metrics AS m ON m.metricid = r.metricid
```

5.2 Code Structure

Figure 5.1 shows the structure of the code, with the most important files and folders highlighted. The graphic starts with the folder `cave-is-prototype`, which can be found when the repository [40] is cloned.

This folder contains two important files and the `src` folder. The `docker-compose.yaml` file is responsible for starting essential components such as the MySQL instance and Grafana, which is described in Subsection 5.1.2. The `.env` file is used to define all secrets used in the code, such as tokens and passwords. For security reasons, an `.env.example` file with non-value keys is provided to be used when the repository is cloned. To use it, one must create an `.env` file containing the same keys as the `.env.example` file, while also adding the appropriate values.

The `src` folder contains two files and four subfolders. These subfolders are structured according to the components shown in Figure 4.1, which are explained in detail in Section 4.2. The files `metricvisualizer.py` and `util.py` are also included. The `metricvisualizer.py` file is the main file of the project. Executing this file triggers the main function. The `util.py` file contains utility functions that are required for the CAVE tool, such as loading configuration files and environment variables that are defined in the `.env` file.

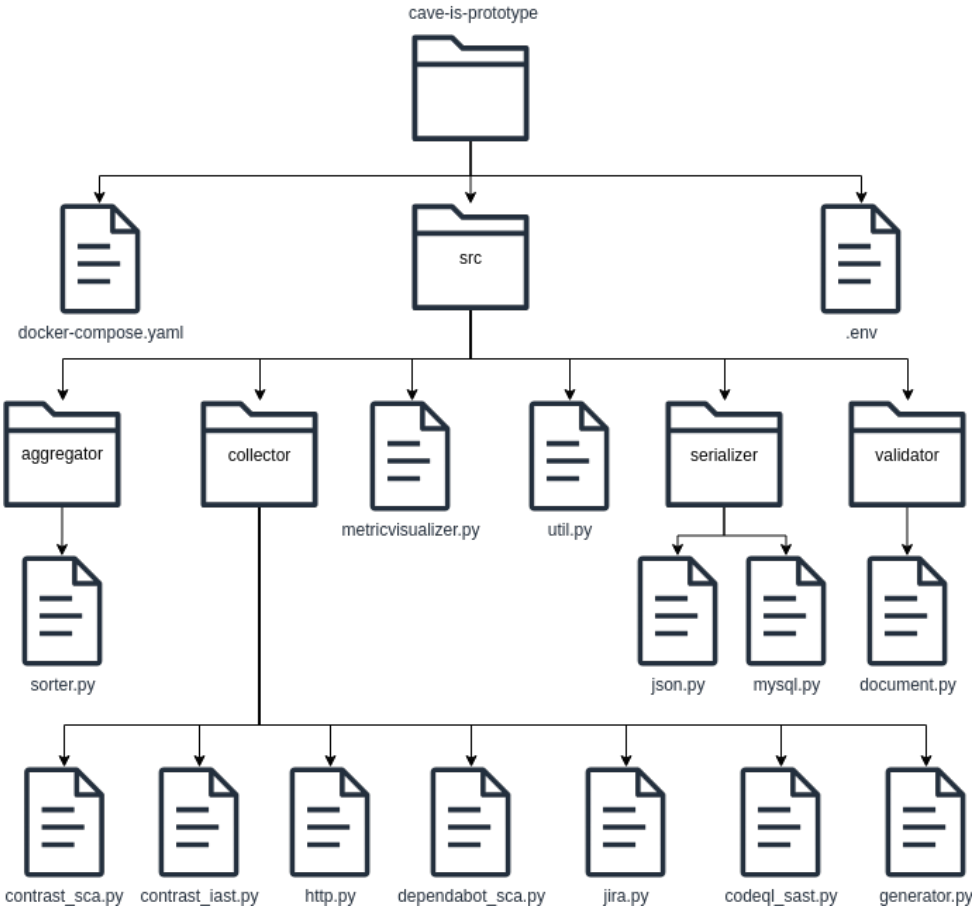

Figure 5.1: Code Structure of CAVE

Chapter 6

Evaluation

The focus of this independent study is on the implementation of a prototype for the automatic collection and analysis of metrics related to DevSecOps activities. This chapter is dedicated to evaluating the effectiveness and functionality of the prototype. First of all, it describes the evaluation methodology used. The detailed evaluation process and results are then presented in Section 6.2. The chapter concludes with a discussion that highlights the strengths and weaknesses of the prototype and provides a comprehensive evaluation of its effectiveness.

6.1 Methodology

The prototype is evaluated using a use case analysis, where performance and functionality are assessed using DevSecOps activities. This approach helps to observe behavior, identify issues, and evaluate the effectiveness of collecting and analyzing metrics. A use case analysis is a methodology for identifying, clarifying, and organizing system requirements [49]. It is a sequence of interactions between systems and users to achieve specific goals. Each use case contains three essential elements: the actor (the system user), the goal (the successful outcome that completes the process), and the system (the steps and functional requirements needed to achieve the end goal).

In this use case analysis, it is assumed that the actor is a security engineer with expertise in DevSecOps and general technical topics. This assumption is based on the design of the prototype, which is intended for use by security professionals and not the general public. The security engineer is pictured as the person responsible for DevSecOps in a company or organization with the knowledge to define relevant activities and the ability to define typical metrics to evaluate these activities.

6.2 Execution

Create Main Configuration File: Creating the configuration file is the most important task of the prototype. Since the configuration is compiled into a single file, it can become quite large. To handle this complexity, the *include* tag can be used to split the configuration into multiple files, which will be merged at the beginning of the CAVE program.

In the main configuration file, there are three important sections to define: items, metrics, and serializer (see Figure 6.1). This modular approach helps to organize and manage the configuration.

```
{
    "items": {
        "APP-1": {
            "include": "example/items/APP-1.json"
        },
        "APP-2": {
            "include": "example/items/APP-2.json"
        }
    },
    "metrics": {
        "include": "example/metrics/metrics.json"
    },
    "serializer": {
        "type": "mysql",
        "host": "127.0.0.1",
        "database": "caveDB",
        "port": 3306,
        "output": "debug.json",
        "include": "example/auth-mysql.json"
    }
}
```

Figure 6.1: Main Configuration File

Configuring Metrics: First, all important DevSecOps activities must be configured as shown in Figure 6.2. Metrics must then be defined to determine whether an activity was successful or not. This is done by configuring the *validator* and *aggregator*. While configuring the metric, the *validator* must be set up to determine the criteria for passing or failing data. In addition, the *aggregator* must be configured to handle missing data and define the required number of items that must be passed for the entire activity to be considered successful. An example configuration is shown in Figure 6.3, where the SAST-L1 *validator* and *aggregator* are configured. The configuration specifies that the activity will fail if there is one open vulnerability with critical or high severity.

```
{
    "threat_modelling-L1": {
        "include": "example/metrics/threatModel/threatModel-L1.json"
    },
    "sca-L1": {
        "include": "example/metrics/sca/sca-L1.json"
    },
    "sast-L1": {
        "include": "example/metrics/sast/sast-L1.json"
    },
    "security_issue_tracking_desing-L1": {
        "include": "example/metrics/securityIssueTrackingDesign/securityIssueTrackingDesign-L1.json"
    }
}
```

Figure 6.2: Configuration of all DevSecOps Activities

```
{
    "level": "1",
    "validator": [
        {
            "result": "fail",
            "prio": 2,
            "description": "Critical vulnerability is present",
            "str:response.state": "open",
            "str:response.rule.security_severity_level": "critical|high"
        },
        {
            "result": "pass",
            "prio": 3,
            "description": "No critical vulnerability"
        }
    ],
    "aggregator": {
        "missing": {"result": "pass", "description": "CodeQL found no vulnerability"}
    }
}
```

Figure 6.3: SAST-L1 Configuration for CodeQL

Configuring Items: When configuring items, it is important to know that each item runs through all predefined metrics and initially fails. To prevent an item from failing, it is necessary to define its activity with a *collector*. The *collector* is responsible for collecting the required data. As shown in Figure 6.4, the activity sast-L1 contains the *collector* codeql_sast-collector, which is configured separately with the include key (Figure 6.5). An important key in the *collector* configuration is the type key, as the CAVE tool uses this as the basis for selecting the *collector* to be used. It can also be observed that the base key is overwritten in the configuration so that the user can define specific parameters, such as a URL for the *collector*.

```
{
    "threat_modelling-L1": {
        "collector": {
            "include": "example/collector/http-collector.json",
            "url": "https://sumsumcity.atlassian.net/wiki/spaces/~7120207e7efd6d002e4139b588c02be762c850/pages/622603/Threat+Model+APP-2"
        }
    },
    "sca-L1": {
        "collector": {
            "include": "example/collector/dependabot_sca-collector.json",
            "base": "https://api.github.com/repos/sumsumcity/cave"
        }
    },
    "sast-L1": {
        "collector": {
            "include": "example/collector/codeql_sast-collector.json",
            "base": "https://api.github.com/repos/sumsumcity/cave"
        }
    },
    "security_issue_tracking_desing-L1": {
        "collector": {
            "include": "example/collector/jira-collector.json"
        }
    }
}
```

Figure 6.4: Item Configuration

```
{
    "type": "codeql_sast",
    "base": "https://api.github.com/repos/sumsumcity/webgoatOutdated",
    "params": {"per_page": 5000},
    "verify": false,
    "headers": {
        "Authorization": "Bearer ${GITHUB_PAT}"
    }

}
```

Figure 6.5: CodeQL Collector Configuration

Setup Database: When setting up MySQL, note that the CAVE tool only fills in the results table automatically. This table must have exactly the format shown in Figure 6.7, with the following columns: itemid, metricid, status, description, realDate, and effectiveDate. In addition, itemid and metricid must be foreign keys that refer to the itemid and metricid columns in the items and metrics tables. Figure 6.6 illustrates the setup of the items table and the definition of the itemid column. The user can decide which additional columns are to be included in the items and metrics tables.

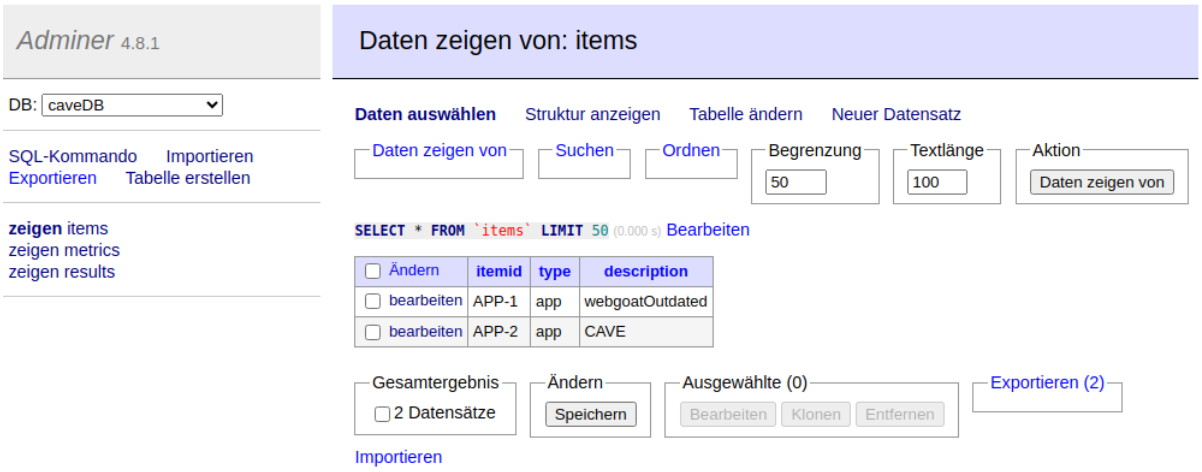

Figure 6.6: Items Table in MySQL

Run CAVE Tool: To run the CAVE tool, ensure that all configurations are set up and the database is configured correctly to avoid errors during execution. The prototype is executed using the metricvisualizer script with the main configuration file as a parameter. The command is as follows:

```
python src/metricvisualizer.py example/config.json
```

If everything has been completed correctly, the status of each DevSecOps activity for all items is saved in the results table of the database, as shown in Figure 6.7.

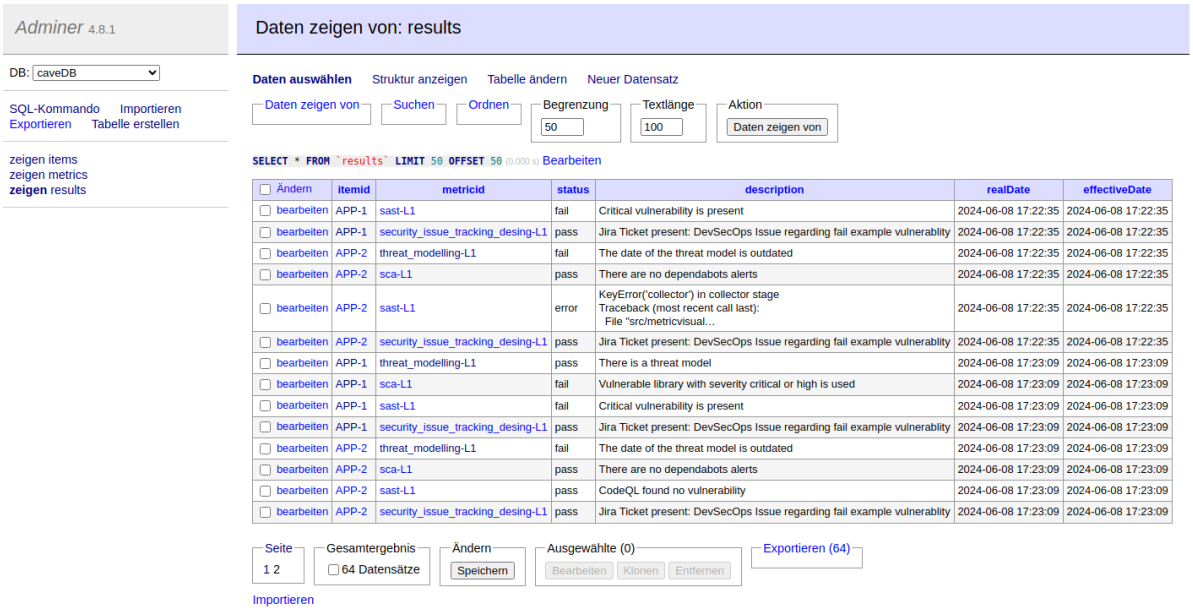

Figure 6.7: Results Table in MySQL

Setup Grafana: To visualize the gathered data, connect the database to Grafana by adding a new MySQL data source (Connections - Add new connections). Once the connection

is established, the database can be queried via the Grafana GUI to obtain the desired data. After retrieving the data, various transformations can be performed, as shown in Figure 6.8. In the example shown in Figure 6.9, the ”Grouping to Matrix” transformation was used, where the columns of the matrix contain the metricid, the rows contain the itemid, and the cell values contain the details that concatenate the status and description from the database. For a colorful representation, as shown in Figure 6.9, value mappings can be added in the Grafana GUI that allow specifying regex and corresponding colors.

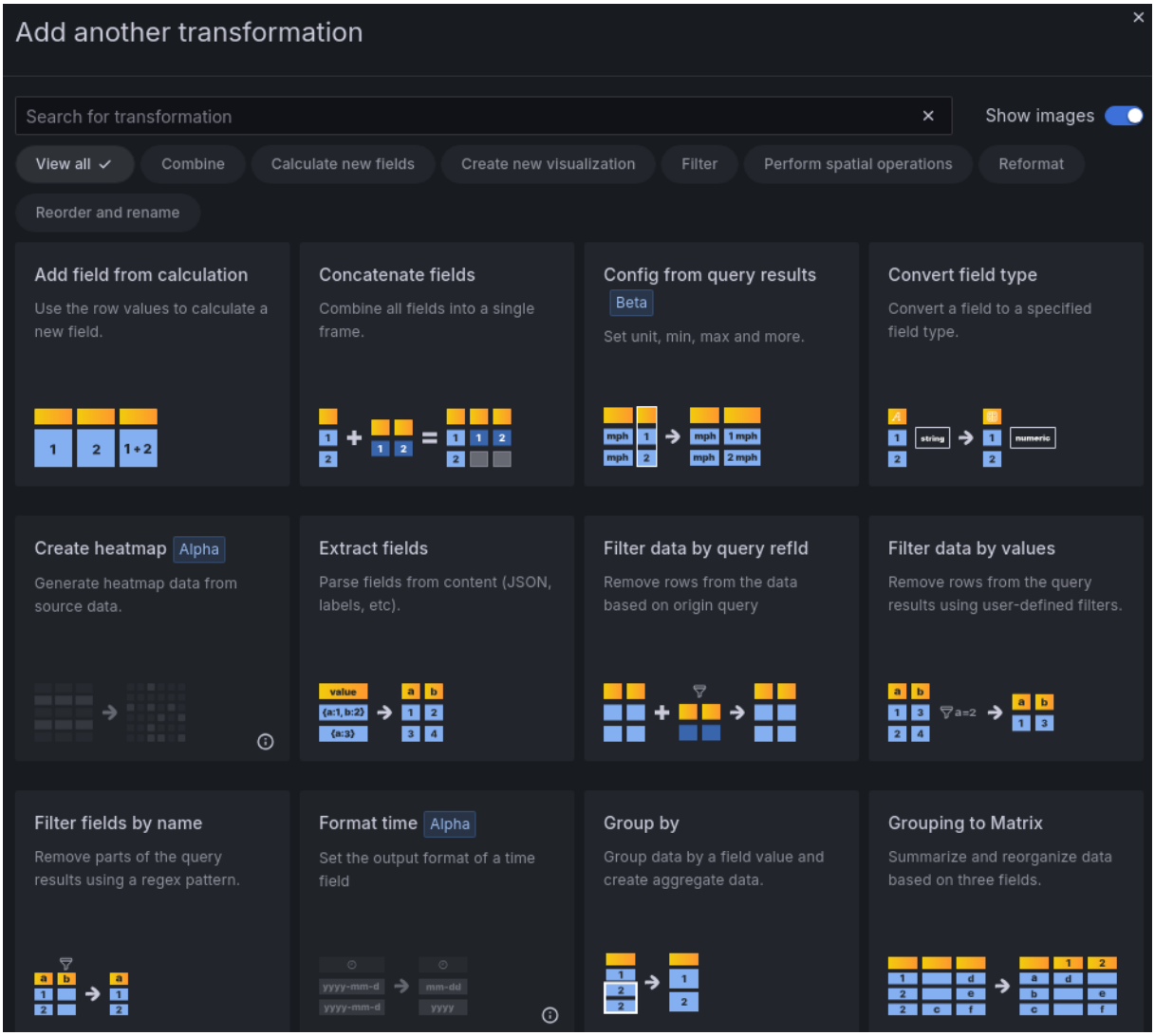

Figure 6.8: Data Transformation Options in Grafana

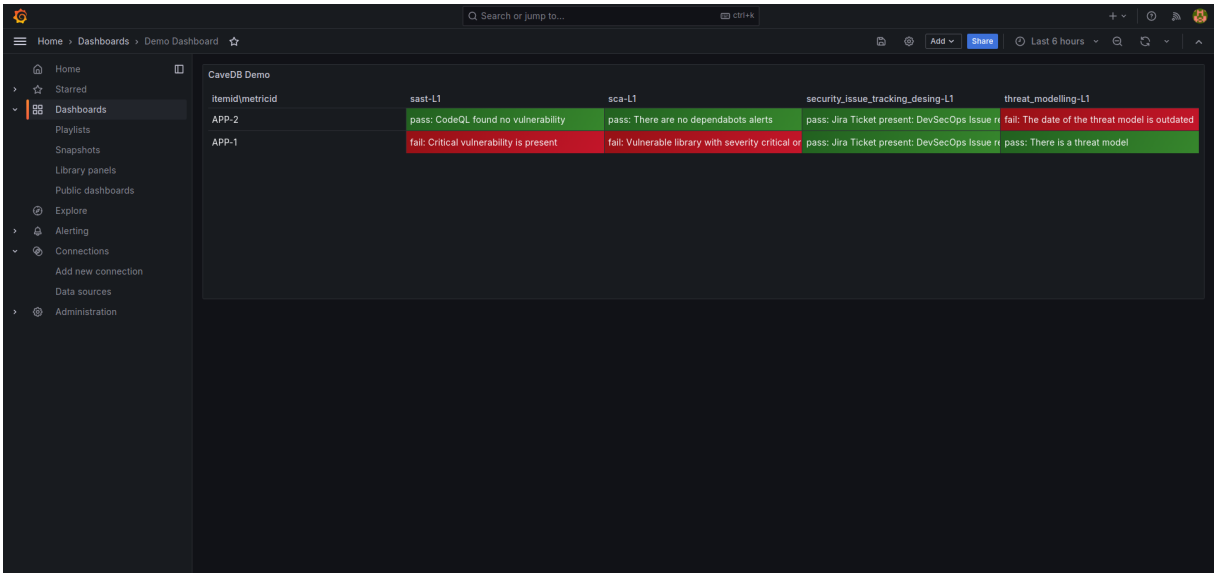

Figure 6.9: Example Grafana Dashboard

6.3 Discussion

To conclude the evaluation, a number of key points about the prototype should be discussed. One of its main strengths is its high configurability. The prototype allows adding numerous items, activities and metrics, providing flexibility to meet different requirements. Its architecture allows for the simple integration of additional technologies with a focus on configuration.

The prototype efficiently collects and analyzes the metrics defined in the Table 4.2. Metrics defined in the table are analyzed according to user requirements. In addition, the tools provided offer extensive possibilities. For example, the HTTP Collector can collect data from various sources and thus enables defining DevSecOps activities that go beyond those listed in Table 4.2.

Another notable strength is the seamless integration with Grafana, which enables effective data visualization. Users have the freedom to customize the way data is presented, which increases the overall usability of the prototype. To summarize, the prototype provides users with great flexibility to define and manage their DevSecOps activities as they desire.

While the configurability of the prototype is its biggest strength, it also leads to complexity and potential problems. One such problem is the lack of clarity about where data should be filtered. In Figure 6.3, the *validator* checks the status and severity. However, these elements could also be filtered in the parameters of the *collector*. This duality can lead to confusion as the *collector* could be misused as an *analyzer* which could lead to problems later on.

Additionally, as shown in Figure 6.3, the *validator* relies on the API response to analyze the data. This dependency complicates using the prototype as users need to understand how each API responds to requests.

Another issue identified during the evaluation is the confusion between DevSecOps activities and metrics within the prototype. Currently, the *validator* validates the metrics of an activity. Due to the above mentioned issue in the *validator*, it can only check the API response key, which leads to the limitation that each activity can only have one *collector* to retrieve the data.

Finally, there is a problem with elements that go through all activities and initially fail if there is no data for a specific metric. The term items suggests that it could also refer to entities other than applications. However, the prototype currently only supports applications, as most metrics are designed for them. For example, if a team needs to be evaluated, it should be measured based on activities and metrics that are specific to teams, not metrics that apply to applications.

As described in Section 3.3, there is limited literature on the practical implementation of tools for automatically collecting and visualizing DevSecOps metrics. The aim of this PoC is to fill this gap by proposing a potential solution, as the literature review in Chapter 3 highlighted the need for such a tool. In addition, this independent study aimed to identify and implement the most important requirements for such a tool, where the findings from the literature research served as a basis.

Chapter 7

Summary and Conclusions

The final chapter of this independent study summarizes the different phases of the study, highlights the key findings and makes suggestions for future research and development. It begins with a general overview of the work and the specific methods used and draws conclusions at the end. Finally, possible areas for future work are outlined.

This work aimed to automate the collection and analysis of DevSecOps metrics by researching, designing and implementing a prototype. The aim was to develop an initial prototype that would enable automated data collection and analysis for a number of key metrics identified through a literature review.

The project began with a comprehensive literature review of the current landscape of DevSecOps challenges, metrics and tools. This research highlighted the need for a flexible, configurable tool that could be adapted to different DevSecOps environments. Based on these findings, the CAVE tool (Collect Analyze Visualize Empower) was designed and developed. The CAVE tool was designed to support multiple technologies and metrics, with a focus on configurability. The prototype was implemented with a modular architecture that facilitated adding new *collectors*. The implementation also included integration with Grafana for advanced data visualization. The methodology used to evaluate the prototype included a use case analysis. The evaluation demonstrated the flexibility of the tool and highlighted areas for improvement.

In conclusion, the CAVE prototype represents a major step forward in automating the collection and analysis of DevSecOps metrics. While there are areas for improvement, the tool's flexible and modular design provides a solid foundation for future development and integration into different DevSecOps environments.

7.1 Future Work

This section outlines approaches for improving the current prototype and adding new features, addressing the issues mentioned in Section 6.3 and suggesting ideas for future improvements.

Many of the problems discussed in Section 6.3 come from the absence of predefined documents in the prototype. One solution is for *collectors* to retrieve data from the API and convert it into predefined document structures, similar to classes. This approach would resolve three of the four issues highlighted in Section 6.3. Firstly, it would standardize data storage and eliminate confusion regarding where to apply filters, which would then be restricted to the *validator*. Secondly, by utilizing predefined documents, the *validator* would no longer depend on API responses, as the necessary keys would be standardized. The *collector* would handle API interactions and store the response data in the predefined document format. Lastly, this solution would enable support for multiple technologies for a single activity and its metrics, as the *collector* would ensure data is stored in a standardized manner.

In addition to these structural changes, an important aspect is the handling of data within the system, especially if there are predefined documents that facilitate integration into the code. As depicted in Figure 6.7, there are currently two types of dates: realDate and effectiveDate, both of which are filled with the same date. However, a planned functionality for the CAVE tool includes the ability to predict when an activity might fail in the future. This information will be displayed in Grafana and the owner of the application will be notified.

After all, the project's aim is to support a broad range of technologies, given the numerous security tools available for generating DevSecOps metrics. The prototype's architecture facilitates the addition of further *collectors* without significant challenges. It is essential that these technologies can be utilized in diverse ways. For instance, Contrast Security uses an integrated scoring system to identify critical vulnerabilities in libraries. This capability could be extended beyond third-party libraries for Software Composition Analysis (SCA) to include patch management activities. Therefore, *collectors* should be customizable to include tool-specific features and capabilities. Additionally, it is important to broaden the serialization options. Currently, the CAVE tool can produce JSON output and store data in a MySQL database. However, to enable a wider range of tools for storing collected and analyzed data, the implementation of additional serialization formats is essential.

Bibliography

[1] H. Myrbakken and R. Colomo-Palacios, "DevSecOps: A multivocal literature review", Sep. 9, 2017, pp. 17–29, ISBN: 978-3-319-67382-0. DOI: 10.1007/978-3-319-67383-7_2.

[2] H. P. Enterprise, "Application security and devops", Technical report, Hewlett Packard Enterprise, Tech. Rep., 2016.

[3] "What is DevSecOps?", IBM. (Oct. 6, 2021), [Online]. Available: https://www.ibm.com/topics/devsecops (visited on 06/01/2024).

[4] M. Dawson, D. Burrell, E. Rahim, and S. Brewster, "Integrating software assurance into the software development life cycle (SDLC)", *Journal of Information Systems Technology and Planning*, vol. 3, pp. 49–53, Jan. 1, 2010.

[5] "OWASP DevSecOps verification standard | OWASP foundation", OWASP. (), [Online]. Available: https://owasp.org/www-project-devsecops-verification-standard/ (visited on 06/08/2024).

[6] A. Jerbi. "KPIs for managing and optimizing devsecops success", InfoWorld. (Nov. 13, 2017), [Online]. Available: https://www.infoworld.com/article/3237046/kpis-for-managing-and-optimizing-devsecops-success.html (visited on 06/01/2024).

[7] V. Casola, A. De Benedictis, M. Rak, and V. Umberto, "A security metric catalogue for cloud applications", Jul. 1, 2018, pp. 854–863, ISBN: 978-3-319-61565-3. DOI: 10.1007/978-3-319-61566-0_81.

[8] N. Medeiros, N. Ivaki, P. Costa, and M. Vieira, "Software metrics as indicators of security vulnerabilities", in *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, ISSN: 2332-6549, Oct. 2017, pp. 216–227. DOI: 10.1109/ISSRE.2017.11. [Online]. Available: https://ieeexplore.ieee.org/document/8109088 (visited on 06/01/2024).

[9] H. Zo, D. L. Nazareth, and H. K. Jain, "Security and performance in service-oriented applications: Trading off competing objectives", *Decision Support Systems*, vol. 50, no. 1, pp. 336–346, Dec. 1, 2010, ISSN: 0167-9236. DOI: 10.1016/j.dss.2010.09.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167923610001715 (visited on 06/01/2024).

[10] W. Mallouli, A. Cavalli, A. Bagnato, and E. Montes de Oca, "Metrics-driven DevSecOps", Jan. 1, 2020, pp. 228–233. DOI: 10.5220/0009889602280233.

[11] A. Bagnato. "ITEA 4 · project · 14009 MEASURE", itea4.org. (), [Online]. Available: https://itea4.org/project/measure.html (visited on 06/08/2024).

[12] "Home", GitHub. (Jan. 22, 2019), [Online]. Available: https://github.com/ITEA3-Measure/Measures/wiki/Home (visited on 06/08/2024).

[13] H. Y. William Richard Nichols, C. L. M. Luiz Antunes, and R. McCarthy, "Automated data for DevSecOps programs", Acquisition Research Program, Technical Report, May 2, 2022, Accepted: 2022-05-05T19:18:56Z. [Online]. Available: https://dair.nps.edu/handle/123456789/4584 (visited on 06/01/2024).

[14] J. Díaz, J. Pérez-Martínez, M. López-Peña, G. Mena, and A. Yague, "Self-service cybersecurity monitoring as enabler for DevSecOps", *IEEE Access*, vol. PP, pp. 1–1, Jul. 19, 2019. DOI: 10.1109/ACCESS.2019.2930000.

[15] R. Fujdiak, P. Mlynek, P. Mrnustik, *et al.*, "Managing the secure software development", in *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, ISSN: 2157-4960, Jun. 2019, pp. 1–4. DOI: 10.1109/NTMS.2019.8763845. [Online]. Available: https://ieeexplore.ieee.org/document/8763845 (visited on 06/01/2024).

[16] M. Marandi, A. Bertia, and S. Silas, "Implementing and automating security scanning to a DevSecOps CI/CD pipeline", in *2023 World Conference on Communication & Computing (WCONF)*, Jul. 2023, pp. 1–6. DOI: 10.1109/WCONF58270.2023.10235015. [Online]. Available: https://ieeexplore.ieee.org/document/10235015 (visited on 06/01/2024).

[17] C. Arunachalam. "Secure SDLC and DevSecOps | LinkedIn", Linkedin. (Aug. 13, 2021), [Online]. Available: https://www.linkedin.com/pulse/secure-sdlc-devsecops-chidhanandham-arunachalam/ (visited on 06/01/2024).

[18] N. Tomas, J. Li, and H. Huang, "An empirical study on culture, automation, measurement, and sharing of DevSecOps", in *2019 International Conference on Cyber Security and Protection of Digital Services (Cyber Security)*, Jun. 2019, pp. 1–8. DOI: 10.1109/CyberSecPODS.2019.8884935. [Online]. Available: https://ieeexplore.ieee.org/document/8884935 (visited on 06/01/2024).

[19] R. Mao, H. Zhang, Q. Dai, *et al.*, "Preliminary findings about DevSecOps from grey literature", in *2020 IEEE 20th International Conference on Software Quality, Reliability and Security (QRS)*, Dec. 2020, pp. 450–457. DOI: 10.1109/QRS51102.2020.00064. [Online]. Available: https://ieeexplore.ieee.org/document/9282798 (visited on 06/01/2024).

[20] H. Yasar, *Overcoming DevSecOps challenges*, Apr. 17, 2020. [Online]. Available: https://apps.dtic.mil/sti/pdfs/AD1110359.pdf.

[21] A. A. U. Rahman and L. Williams, "Software security in DevOps: Synthesizing practitioners' perceptions and practices", in *2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED)*, May 2016, pp. 70–76. [Online]. Available: https://ieeexplore.ieee.org/document/7809439 (visited on 06/01/2024).

[22] Z. Ahmed and S. Francis, "Integrating security with DevSecOps: Techniques and challenges", Nov. 1, 2019, pp. 178–182. DOI: 10.1109/ICD47981.2019.9105789.

[23] L. Singla. "What is DevSecOps? definition, challenges, and best practices", Net Solutions. (Sep. 22, 2022), [Online]. Available: https://www.netsolutions.com/insights/what-is-devsecops/ (visited on 06/01/2024).

[24] S. Rafi, W. Yu, M. Azeem Akbar, A. Alsanad, and A. Gumaei, "Prioritization based taxonomy of DevOps security challenges using PROMETHEE", *IEEE Access*, vol. PP, pp. 1–1, Jun. 1, 2020. DOI: 10.1109/ACCESS.2020.2998819.

[25] X. Zhou, R. Mao, H. Zhang, *et al.*, "Revisit security in the era of DevOps: An evidence-based inquiry into DevSecOps industry", *IET Software*, vol. 17, n/a–n/a, Jul. 26, 2023. DOI: 10.1049/sfw2.12132.

[26] R. Rajapakse, M. Zahedi, M. Ali Babar, and H. Shen, "Challenges and solutions when adopting DevSecOps: A systematic review", *Information and Software Technology*, vol. 141, p. 106 700, Aug. 1, 2021. DOI: 10.1016/j.infsof.2021.106700.

[27] R. Desai and T. N. Nisha, "Best practices for ensuring security in DevOps: A case study approach", *Journal of Physics: Conference Series*, vol. 1964, no. 4, p. 042 045, Jul. 2021, Publisher: IOP Publishing, ISSN: 1742-6596. DOI: 10.1088/1742-6596/1964/4/042045. [Online]. Available: https://dx.doi.org/10.1088/1742-6596/1964/4/042045 (visited on 06/01/2024).

[28] M. A. Akbar, K. Smolander, S. Mahmood, and A. Alsanad, "Toward successful DevSecOps in software development organizations: A decision-making framework", *Information and Software Technology*, vol. 147, p. 106 894, Jul. 1, 2022, ISSN: 0950-5849. DOI: 10.1016/j.infsof.2022.106894. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0950584922000568 (visited on 06/01/2024).

[29] U. G. S. Administration. "DevSecOps guide", Tech at GSA. (), [Online]. Available: https://tech.gsa.gov/guides/dev_sec_ops_guide/ (visited on 06/01/2024).

[30] R. Amaro, R. Pereira, and M. Mira da Silva, "DevOps metrics and KPIs: A multivocal literature review", *ACM Computing Surveys*, vol. 56, no. 9, 231:1–231:41, Apr. 25, 2024, ISSN: 0360-0300. DOI: 10.1145/3652508. [Online]. Available: https://dl.acm.org/doi/10.1145/3652508 (visited on 06/01/2024).

[31] M. Thevarmannil. "DevSecOps metrics & KPIs for 2024", Practical DevSecOps. (Jan. 11, 2024), [Online]. Available: https://www.practical-devsecops.com/devsecops-metrics/ (visited on 06/01/2024).

[32] B. Nichols. "The current state of DevSecOps metrics", Software Engineering Institute Carnegie Mellon University. (Mar. 29, 2021), [Online]. Available: https://insights.sei.cmu.edu/blog/the-current-state-of-devsecops-metrics/ (visited on 06/01/2024).

[33] A. Crouch. "DevSecOps: Incorporate security into DevOps to reduce software risk", AgileConnection. (Dec. 13, 2017), [Online]. Available: https://www.agileconnection.com/article/devsecops-incorporate-security-devops-reduce-software-risk (visited on 06/01/2024).

[34] F. José. "Effective DevSecOps", Medium. (Jul. 26, 2018), [Online]. Available: https://medium.com/@fabiojose/effective-devsecops-f22dd023c5cd (visited on 06/01/2024).

[35] A. Schurr. "Mobile app DevOps metrics that matter - NowSecure", NowSecure. (Feb. 27, 2019), [Online]. Available: https://www.nowsecure.com/blog/2019/02/27/mobile-app-devops-metrics-that-matter/ (visited on 06/01/2024).

[36] T. Longstaff and H. Yasar, "Build secure application with DevSecOps!", 2018. [Online]. Available: https://apps.dtic.mil/sti/tr/pdf/AD1088677.pdf.

[37] L. Prates, J. Faustino, M. Silva, and R. Pereira, "DevSecOps metrics", in Aug. 8, 2019, pp. 77–90, ISBN: 978-3-030-29607-0. DOI: 10.1007/978-3-030-29608-7_7.

[38] E. Chickowski. "Seven winning DevSecOps metrics security should track", Bitdefender Blog. (May 1, 2018), [Online]. Available: https://www.bitdefender.com/blog/businessinsights/seven-winning-devsecops-metrics-security-should-track/ (visited on 06/01/2024).

[39] "FrontPage - python wiki", Python. (), [Online]. Available: https://wiki.python.org/moin/ (visited on 06/09/2024).

[40] R. Wäspi, *Sumsumcity/cave-is-prototype*, original-date: 2024-05-11T10:51:57Z, Jun. 12, 2024. [Online]. Available: https://github.com/sumsumcity/cave-is-prototype (visited on 06/14/2024).

[41] *Requests: Python HTTP for humans.* Version 2.32.3. [Online]. Available: https://requests.readthedocs.io (visited on 06/09/2024).

[42] *Python-dotenv: Read key-value pairs from a .env file and set them as environment variables*, version 1.0.1. [Online]. Available: https://github.com/theskumar/python-dotenv (visited on 06/09/2024).

[43] *Mysqlclient: Python interface to MySQL*, version 2.2.4.

[44] *Isodate: An ISO 8601 date/time/duration parser and formatter*, version 0.6.1. [Online]. Available: https://github.com/gweis/isodate/ (visited on 06/09/2024).

[45] "Docker overview", Docker Documentation. (700), [Online]. Available: https://docs.docker.com/guides/docker-overview/ (visited on 06/14/2024).

[46] "MySQL :: MySQL 8.4 reference manual :: 1.2.1 what is MySQL?" (), [Online]. Available: https://dev.mysql.com/doc/refman/8.4/en/what-is-mysql.html (visited on 06/14/2024).

[47] "About grafana | grafana documentation", Grafana Labs. (), [Online]. Available: https://grafana.com/docs/grafana/latest/introduction/ (visited on 06/14/2024).

[48] "Data sources | grafana documentation", Grafana Labs. (), [Online]. Available: https://grafana.com/docs/grafana/latest/datasources/ (visited on 06/14/2024).

[49] K. Brush. "What is a use case?", TechTarget. (2020), [Online]. Available: https://www.techtarget.com/searchsoftwarequality/definition/use-case (visited on 06/15/2024).

Abbreviations

CD	Continuous Delivery
CI	Continuous Integration
HTTP	Hypertext Transfer Protocol
IAST	Interactive Application Security Testing
JSON	JavaScript Object Notation
MLT	Mean Lead-Time
MTTD	Mean Time to Detect
MTTR	Mean Time to Resolve
OWASP	Open Web Application Security Project
PoC	Proof of Concept
SaaS	Software as a Service
SAST	Static Application Security Testing
SCA	Software Composition Analysis
SDLC	Software Development Life Cycle
SMM	Structured Metrics Metamodel
SSDLC	Secure Software Development Life Cycle

Glossary

Agile A software development methodology that emphasizes iterative progress, collaboration and flexibility to adapt to changing requirements.

Continuous Delivery (CD) A practice in software development in which code changes are automatically prepared for production release.

Continuous Integration (CI) A development practice in which developers often merge their code changes in a central repository, followed by automated builds and tests.

Defect Burn Rate A metric that indicates how often vulnerabilities are introduced into production.

DevOps A set of practices that combine software development (Dev) and operations (Ops) to shorten the development cycle and continuously deliver high-quality software.

DevSecOps A methodology that integrates security practices into the DevOps process and ensures that security is a shared responsibility throughout the software development lifecycle.

Environment Variables Variables that are defined in the operating system and used by applications to configure settings such as paths, authentication tokens and configuration parameters.

Graphical User Interface (GUI) Graphical user interface of software that facilitates the interaction with an application.

ISO 8601 An international standard for the presentation of date and time-related data.

JSON Data transfer format for the exchange of data between applications in an easily readable text form.

Metric A measurement standard.

Modules Reusable code components in Python that can be imported and used in different parts of a program.

MLT (Mean Lead-Time) The average time between transferring the code to production and deployment.

MTTD (Mean Time to Detect) The average time it takes to detect a vulnerability from the time it occurs.

MTTR (Mean Time to Resolve) The average time it takes to fix a vulnerability after it has been identified.

MySQL An open-source relational database management system used for storing and managing data.

Number of Deployments A metric that indicates how often code changes are used in production.

Number of Failed Deployments A metric that counts the number of failed deployments.

Number of Security-related Tickets A metric that counts the number of tickets flagged with security concerns.

Number of Security Vulnerabilities A metric that counts the total number of vulnerabilities present in an application.

Patch Management The process of managing software updates for operating systems, applications and devices to close security gaps.

Prototype An early sample or model of a product built to test a concept or process.

Shift-Left A principle in software development in which security and testing activities are carried out at an earlier stage of the development process.

Test Coverage A metric that measures the percentage of code covered by automated tests.

List of Figures

List of Tables

Appendix A

Installation Guidelines

All source code developed as part of this thesis can be found on GitHub [40]. A README is provided for each package, which always includes instructions for installation and local deployment. There is also a Docker Compose script in the infrastructure repository that simplifies running a demonstration of the prototype. To run this demonstration, follow the instructions in the README of this repository.