An Empirical Study of DevSecOps Focused on Continuous Security Testing

Clarisse Feio
INOV / INESC-ID / IST
Lisbon, Portugal
c.feio@tecnico.ulisboa.pt

Nuno Santos
INESC-ID / IST
Lisbon, Portugal
nuno.m.santos@tecnico.ulisboa.pt

Nelson Escravana
INOV
Lisbon, Portugal
nelson.escravana@inov.pt

Bernardo Pacheco
INOV
Lisbon, Portugal
bernardo.pacheco@inov.pt

***Abstract*—DevSecOps is an emerging approach to integrate robust security into the DevOps software development process. It focuses on breaking the silos between development, security, and operations and on introducing security from the beginning of the software development process. In this paper, we present a DevSecOps framework centered on the principle of continuous security testing, applicable across various software development scenarios. Our ultimate goal is to promote wider adoption of DevSecOps practices. The framework comprises a CI/CD pipeline, a series of activities tailored for each phase, and tools to automate these activities. Through a case study conducted in a real-world setting, we evaluated the effectiveness of our framework. The results indicate that the framework's implementation was successful, enabling the development team to identify numerous vulnerabilities, including critical ones, proactively. Moreover, the developers have shown a keen interest in employing this framework in their future projects.**

1. Introduction

Software development has evolved significantly over the years, driven by an increasing demand for speed and shorter time to market. This evolution has led to the emergence of agile methodologies such as SCRUM, Kanban, Extreme Programming (XP), and DevOps [10], all aimed at reducing development time so products can swiftly reach their final clients [10, 16]. DevOps, in particular, stands out as the most recent among these methodologies and has been widely adopted by numerous organisations [9]. Its name, derived from the words "Development" and "Operations", highlights its focus on fostering collaboration between these traditionally separate silos.

Evolving from DevOps [13, 17], DevSecOps aims additionally to prevent cyberattacks and mitigating their potentially devastating consequences for organisations [14, 17]. It promotes collaboration not only between development and operations teams, but also includes security teams in the process [13, 15, 17]. Specifically, DevSecOps aims to integrate security measures and automate security practices from the project's inception, adopting a "shift-left" approach. This strategy ensures that security-related activities are not postponed until the end of the Software Development Lifecycle (SDLC) [7, 9, 17], thereby maintaining the desired speed and agility [10, 13, 17].

Despite the growing popularity of DevSecOps, its adoption is still not as widespread as that of its predecessor, DevOps, which remains the preferred methodology. Several factors contribute to this situation, with the primary concern being the perception of security as a bottleneck that slows down the speed and agility inherent to DevOps processes [6, 13]. Often, developers do not see security as part of their responsibilities, relegating it to the domain of specialised security teams [17]. The difficulty in presenting security as a worthwhile investment also poses a challenge. Security measures can be costly, seen as providing no direct returns, and offer no absolute guarantees of effectiveness.

Considering such existing concerns, this paper aims to demystify the practical application of the DevSecOps methodology in a real-world project, evaluating its tangible benefits for both the software development team and the organisation at large. To this end, we propose to address four main research questions:

- **RQ1. What are the phases of a typical DevSecOps pipeline?** Considering the variety of DevSecOps methodologies proposed in the literature, each with its own advantages and disadvantages, it is crucial to identify an optimal DevSecOps pipeline; our goal is to pinpoint one that incorporates the most beneficial features to facilitate its adoption by organisations.
- **RQ2. What continuous security testing activities are performed in each phase?** By identifying the key activities within each phase of the pipeline, developers can accurately understand the security-related tasks expected of them throughout the development process.
- **RQ3. Which tools are used for continuous security testing?** Many tools are available to assist developers at various stages; therefore, identifying effective and user-friendly tools can help maximise the benefits of continuous security testing for different activities.
- **RQ4. What is the impact of adopting DevSecOps and continuous security testing in software development?** After addressing the preceding questions, it is central to examine the practical implications of these methodologies in a real-world development setting.

By analysing these questions, this paper makes several contributions, each being covered in central sections of the paper. First, in §3, we present a review of the existing literature to explore various DevSecOps pipelines. The objective is to identify commonalities among them and understand their respective advantages and disadvantages, thereby pinpointing an appropriate pipeline for our study and addressing RQ1. Then, drawing from the existing literature, we concentrate on RQ2 by pinpointing the typical activities conducted at each stage of the DevSecOps pipeline (see §4). These activities are then scrutinised to isolate those pertinent to continuous security testing, which are subsequently compiled into a comprehensive

DOI 10.1109/EuroSPW61312.2024.00074

table. Thirdly, §5 addresses RQ3 by examining the types of tools suitable for each security testing activity. We defined three criteria for selecting tools: they must be open source, regularly updated, and recommended by a reputable cybersecurity entity. Based on these criteria, a total of 15 tools were chosen. Lastly, §6 showcases the application of the derived framework to a real case study, thereby addressing RQ4. We applied the framework to an existing project that was already adopting DevOps. The pipeline underwent adaptation, incorporating new activities and tools. This section details the outcomes from utilising the security testing tools and captures the developers' perspectives on the modifications. Overall, the integration of this framework yielded positive results, though some limitations were identified and are discussed.

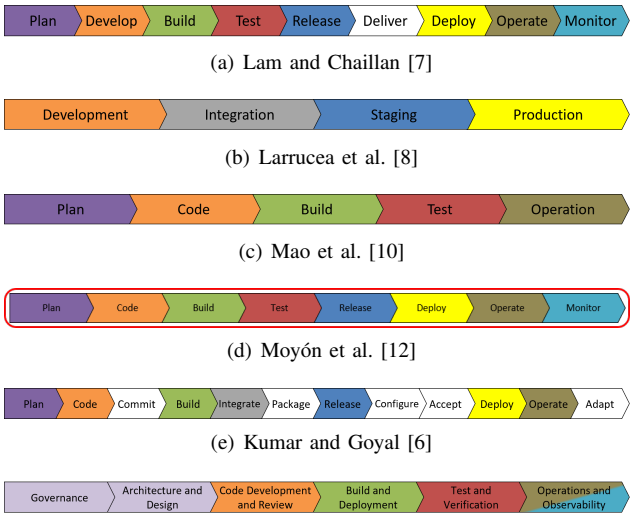
2. Related Work

Although DevSecOps is a relatively new methodology, academic literature on the subject has expanded significantly in recent years. Myrbakken and Colomo-Palacios [13], Sánchez-Gordón and Colomo-Palacios [15], Mao et al. [10], Desai and Nisha [4], Akbar et al. [1], and Leppänen et al. [9] all performed literature reviews to understand the benefits and challenges of adopting DevSecOps. In general, it was found that the main benefits include shifting security to the left and automating security. There were many challenges identified, mainly related to creating a security culture, the lack of appropriate tools, the lack of security specialists, and many others.

Mohan et al. [11], Tomas et al. [17], and Angermeir et al. [2] studied the state of DevSecOps in the software development industry. This was done mainly through interviews with practitioners, which revealed similar results to the literature reviews just mentioned. Namely, the implementation of security activities is still rare given the lack of knowledge of developers, lack of tools and standards, and not perceiving security as their responsibility. Implementing a security culture is a major hurdle, but introducing security champions into the team is beneficial.

From the implementation angle, Lam and Chaillan [7], Larrucea et al. [8], Kumar and Goyal [6], Moyón et al. [12], Rangnau et al. [14], and Brasoveanu et al. [3] deployed DevSecOps in a real context, some also creating their own framework [6, 7, 12]. All of these authors found that practitioners were satisfied with the results and often surprised by the amount of automation possible [12].

In this work, we aim to complement this body of research by ultimately recommending a DevSecOps framework that is both generalisable and applicable across various sectors of the software development industry, thereby facilitating the broader adoption of this methodology. Our investigation predominantly centers on the technical facets of DevSecOps, with a specific emphasis on: the CI/CD pipeline, the principle of continuous security testing, the spectrum of activities integrated throughout the pipeline, and the deployment of tools to streamline automation. Continuous security testing, notable for its capacity for easy automation without impeding speed and agility, can identify a broad range of security flaws and automatically categorise these defects according to their severity. Hence, applying this singular principle to a DevOps project can instantly deliver value to the team, introducing minimal



(a) Lam and Chaillan [7]

(b) Larrucea et al. [8]

(c) Mao et al. [10]

(d) Moyón et al. [12]

(e) Kumar and Goyal [6]

(f) Brasoveanu et al. [3]

Figure 1. Common elements among the analysed pipelines.

disruption or alterations to their existing workflow and productivity levels. It achieves this by providing swift feedback on the software's security posture, thereby facilitating the early detection of security vulnerabilities and streamlining task prioritisation.

3. RQ1: DevSecOps Pipeline Selection

The core component of DevSecOps is the CI/CD pipeline, which is composed of a set of consecutive phases that define various activities and tools to aid the automation of software development. This pipeline can have many variations, depending on the level of detail and intended use. In this section, we assemble a DevSecOps pipeline that can be directly applied to the majority of software development contexts with minimal to no adjustments, therefore making its adoption easier. For this reason, it should meet the following criteria:

- the pipeline should be applicable and easily adaptable to various industry sectors willing to use DevSecOps;
- the pipeline should not be simple to the point of omitting crucial information, but it should also not be too complicated, as it might cause organisations to feel reluctant to adopt it;
- the pipeline should not be technology specific.

By studying the different proposals for DevSecOps pipelines found in the literature, it is possible to pinpoint common aspects among them, which help identifying which phases should be present in a general pipeline. Six pipelines were studied and compared, as presented below.

3.1. Common Elements of Reviewed Pipelines

All the pipelines were set side-by-side so that it would be easier to visually identify common elements. Phases with the same (or similar) content were painted with the same colour, while those without common elements were left blank. This result can be seen in Figure 1.

All pipelines have elements in common with others, but some have all their phases covered by this commonality, while others have phases that were not present in any

other pipeline. This may be due to either using a different nomenclature or due to a different vision/approach. Most commonly, blank phases are already part of other, more general, phases and do not need to be explicitly present in the pipeline. Overall, the phases that are common to all, or the majority, of pipelines are: Plan, Code/Develop, Build, Test, Release, Deploy, Operate, and Monitor.

3.2. Pipeline Advantages and Disadvantages

After analysing each pipeline and comparing it with others, it is possible to identify the benefits and drawbacks of each of them. A summary table with all the advantages and disadvantages of the pipelines is presented in Table 1.

The pipeline by Lam and Chaillan [7] stands out significantly from others, primarily due to its endorsement by the US Department of Defense, encouraging practitioners to adopt it as a model for their organisations. This pipeline has a good amount of detail, without being overbearing.

The pipeline proposed by Larrucea et al. [8] is specifically designed around the environments encountered in their project, rendering it highly customised for that particular application and potentially less relevant for different contexts. This results in a simplified pipeline that omits critical phases such as Plan, Test, and Operate. Additionally, it diverges from conventional pipeline nomenclature and deviates from the reference design.

Mao et al.'s pipeline [10] has a fair level of detail and bears resemblance to the reference design. It highlights relevant phases but omits others (e.g. Release and Deploy), resulting in a simpler pipeline that might mislead professionals with limited software development experience.

The pipeline utilised by Moyón et al. [12] is detailed and is very similar to the reference design, only lacking the Deliver phase. However, this absence does not omit important activities, since the activities conducted in this phase can be done in the Release phase.

Kumar and Goyal's pipeline [6] is the most detailed with a total of 12 phases. On the one hand, the pipeline is very complete, but on the other hand, it is too complex, especially for people unfamiliar with it, which can make them apprehensive. Also, this pipeline is specifically tailored for a project and therefore not always easily applicable to other situations.

Finally, the pipeline from Brasoveanu et al. [3] is a bit different from the usual pipeline. This is due to it being a maturity assessment framework and not necessarily a software development pipeline (but the authors tried to create it so that it could be easily mapped into a pipeline). This results in less detail than desired, and also in a different nomenclature from what is usually utilised.

3.3. Proposed DevSecOps Pipeline

Following the analysis and comparison of the six pipelines, we can see that the common elements between the majority of them are Plan, Code/Develop, Build, Test, Release, Deploy, Operate and Monitor, and that the pipelines with more benefits are those from Lam and Chaillan [7] (DoD Reference design) and Mao et al. [10].

Therefore, the final pipeline built for this framework is composed of those eight common phases: Plan, Code, Build, Test, Release, Deploy, Operate, and Monitor,

TABLE 1. SUMMARY OF PIPELINES' BENEFITS AND DRAWBACKS.

Pipeline	Benefits	Drawbacks
Lam and Chaillan [7]	Good level of detail Reference design	
Larrucea et al. [8]		Too simple, lacks detail Application specific Different nomenclature
Mao et al. [10]	Similar to reference design	Lacks important phases
Moyón et al. [12]	Good level of detail Similar to reference design	
Kumar and Goyal [6]	Very complete	Too complex Application specific
Brasoveanu et al. [3]		Lacks important phases Different nomenclature

as presented in Figure 1(d), which corresponds to the pipeline presented by Moyón et al. [12], and is also very similar to the pipeline developed by Lam and Chaillan [7]. It strikes a good balance between simplicity and complexity, being explicit enough to not hide any important details but simple enough to not seem too difficult and daunting. This kind of pipeline is often also used in plain DevOps, which makes the adaptation of DevOps to DevSecOps much simpler and easier because fewer changes have to be made to the way a team operates.

4. RQ2: DevSecOps Pipeline Activities

With the pipeline defined, the next step involves specifying the activities to be performed at each phase, guided by established DevOps practices such as continuous monitoring, continuous integration, and continuous delivery. In DevSecOps, a new practice arises, *continuous security*, which involves taking security into account in every phase of the pipeline. In particular, *continuous security testing* involves the security testing of software and its constituents along the whole pipeline. This section further elaborates on the pipeline by detailing the activities conducted in each phase, and identifying the three requirements for these activities as outlined in the following list:

- the activities should be related to security testing;
- continuous security testing implies that security testing is performed continuously along the entire SDLC, therefore, every phase of the defined pipeline should have at least one activity related to security testing;
- security testing activities should include various domains, such as source code testing, network testing, infrastructure testing, and many others.

To identify the security testing activities, we must first identify the typical general activities performed along the whole pipeline. To this end, we analyse the literature and identify common themes among the activities proposed by each author. Then, we further investigate them to pinpoint which of them correspond to "continuous security testing" activities and follow the requisites defined previously.

4.1. Identifying Testing Activities

A summary of all activities found in the literature can be found in Table 2. These activities represent the standard practices typically employed in the adoption of DevSecOps and can be customised to meet the specific requirements of an organisation. Customisation may involve modifying the set of activities by either excluding

TABLE 2. DEVSECOPS ACTIVITIES. TESTING ACTIVITIES ARE IDENTIFIED IN BOLD.

Plan	Code	Build	Test	Release	Deploy	Operate	Monitor
• Project mng. planning • Software requirement analysis • Security requirement analysis • System design • Threat modelling • Risk assessment • Configuration management • **Test planning**	• Define secure dev. guidelines • Enforce secure development guidelines • **Code development** • Secure guidelines verification • Code review • **Static code analysis**	• Compile and package • Build automation • **Unit testing** • **Smoke testing** • **Software composition analysis** • **SAST**	• **Integration testing** • **Fuzz testing** • **DAST** • **IAST** • **Vulnerability scanning**	• **Acceptance testing** • **Performance testing** • Quality assurance • Documentation review • **Vulnerability scanning** • **Penetration testing**	• Infrastructure provisioning • Documentation review • **Vulnerability scanning** • **Penetration testing**	• Update and patch • Infrastructure orchestration • Load balancing • Secrets management • Security incident management • **Red & Blue teams** • Bug bounty	• Performance monitoring • Threat intelligence • Logging and log analysis • **Vulnerability scanning** • Intrusion detection

TABLE 3. CONTINUOUS SECURITY TESTING ACTIVITIES IN EACH PHASE.

Plan	Code	Build	Test	Release	Deploy	Operate	Monitor
• Test planning	• Test development • SAST	• Software composition analysis • SAST	• DAST • IAST • Vulnerability scanning	• Vulnerability scanning • Penetration testing	• Vulnerability scanning • Penetration testing	• Red & blue teams	• Vulnerability scanning

certain tasks or incorporating new ones to align with the organisation's unique needs. Overall, we identified a total of 16 testing activities highlighted in bold in Table 2.

Our process to identify all testing activities began by selecting those that have the word "test" or "testing" in their name. Eleven activities meet this condition: test planning, unit testing, smoke testing, SAST, integration testing, fuzz testing, DAST, IAST, acceptance testing, performance testing, and penetration testing.

Moreover, several activities, despite not explicitly bearing the terms "test" or "testing", are inherently linked to the testing process. Specifically, code development covers not only the creation of the software itself but also involves configuration and test development. Consequently, the sub-activity of test development is pertinent in this context. Similarly, static code analysis, which may include a range of source code examination techniques such as SAST, qualifies as a testing activity. Software composition analysis is another crucial activity, aimed at determining whether third-party libraries are vulnerable and assessing their security for use. Additionally, vulnerability scanning, which is effectively a form of vulnerability testing, entails assessing the software and its components for known vulnerabilities using specially designed inputs. The practices of red & blue teams also align closely with testing: red teams actively seek to compromise the system's security, whereas blue teams evaluate whether the existing defence mechanisms, strategies, and procedures are sufficient.

4.2. Identifying Security Testing Activities

Next, we filter the identified testing activities and pinpoint those related to security. In total, nine different security testing activities were identified (see Table 3).

Following a similar approach as previously described, we initially identify activities explicitly containing the word "security" in their names. These include SAST, IAST, and DAST, where the suffix 'ST' denotes "Security Testing". Vulnerability scanning also clearly falls into this category, as it directly involves the identification and mitigation of security vulnerabilities. Penetration testing, along with red & blue team exercises, is similarly unambiguous. These practices not only align closely with vulnerability scanning by delving deeper into discovering the origins of vulnerabilities but also extend to evaluating the effectiveness of existing defence mechanisms.

Other activities may not be as immediately apparent in their connection to security. Software composition analysis, for example, is a critical security activity. It assesses whether components from third-party or open-source projects contain vulnerabilities that could compromise the system. Test planning, which involves defining strategies and plans for all testing activities, inherently includes planning for security tests. Similarly, test development, which involves creating various tests, encompasses the development of security tests as well. By virtue of including security tests, test development is also an essential component of security testing activities.

4.3. Proposed DevSecOps Activities

Our finalised proposal for continuous security testing activities throughout the pipeline is detailed in Table 3. This proposal meets all the defined prerequisites: (i) all activities are related to security testing; (ii) all pipeline phases have at least one security testing activity, as needed for continuous security testing; and (iii) all these activities test various domains. For example, SAST focuses on the source code, while DAST tests the behaviour of the running application when confronted with invalid inputs. Red & blue teams is the most in-depth activity.

5. RQ3: DevSecOps Security Testing Tools

Automation plays a central role in attaining the agility and speed essential for DevSecOps. This section is ded-

icated to identifying tools that can automate, whether partially or fully, the set of continuous security testing activities outlined in the preceding section. To ensure broader adoption, the DevSecOps framework needs to be user-friendly and accessible, without imposing significant delays or financial burdens on the organisation. Hence, selected tools must satisfy the following specific criteria aimed at minimising delays and costs:

- ideally, Open-Source Software (OSS) should be preferred to prevent vendor lock-in and ensure greater transparency and customisation. OSS is cost-effective, as open-source tools are often free for use, even commercially, depending on their license;
- the tools must be up-to-date to address the latest security threats effectively. They should not be deprecated, and their most recent update should have occurred within the past year;
- it is desirable for tools to be recommended or developed by internationally recognised cybersecurity entities, such as the OWASP Foundation, CISA (Cybersecurity & Infrastructure Security Agency), and NIST. This endorsement is typically an indication that the tool meets high-quality standards and adheres to the best practices established by these entities.

Beyond these criteria, organisations may consider additional factors tailored to their specific use cases. The usability of the tool is crucial, as it enhances the efficiency of experts' work. Ease of installation and configuration is also vital. Furthermore, compatibility with existing tools (e.g., Jenkins) is significant, as it facilitates automation. Most importantly, the selected tools must support all the programming languages utilised by the organisation.

5.1. Proposed DevSecOp Toolset

By reviewing the available literature, we identified over 50 tools for all activities and filtered them based on the tool criteria defined. In the end, we selected 15 tools from the initial 54. These can be seen in Table 4.

Notice that not all activities are represented in Table 4, namely test planning, test development, and IAST, which have no associated tools that meet our prerequisites. There are various reasons for this. In the case of test planning, this activity is very manual, where tools are only auxiliary but not obligatory. For this reason, only one tool was found in the chosen literature (Jira Software), which was not open-source and so was excluded from the final list.

Test development is a broad activity that encompasses multiple kinds of testing beyond security testing. The tools utilised in this activity can be used for different types of testing. Since they are not specific to cybersecurity, they were not mentioned by any of the cybersecurity organisations considered (OWASP, NIST, and CISA), and therefore were excluded from the final list.

IAST has different reasons; it is focused on security and dependent on tools, however, there is a lack of tools overall, and the ones that exist are mainly commercial.

6. RQ4: DevSecOps Case Study

While our DevSecOps framework is now fully outlined, assessing its effectiveness in a real-world scenario is crucial to understand its impact on the development process and quality of the software, particularly in terms of security. In this section, we apply our framework to a case study to assess its advantages and limitations.

TABLE 4. FINAL SELECTION OF CONTINUOUS SECURITY TESTING TOOLS BASED ON THE PRE-DEFINED CRITERIA.

Tool name	Activity	OSS	Updated	Entities
SpotBugs	SAST	Yes	Yes	OWASP; NIST
Clair	SAST	Yes	Yes	OWASP
SonarQube	SAST	Yes (also has a commercial version)	Yes	OWASP; NIST
PMD	SAST	Yes	Yes	OWASP; NIST
Dependency-Check	SCA	Yes	Yes	OWASP
bundler-audit	SCA	Yes	Yes	OWASP
Zed Attack Proxy	DAST; Vulnerability scanning; Penetration testing; Red & Blue teams	Yes	Yes	OWASP; CISA
Wapiti	DAST; Vulnerability scanning; Penetration testing; Red & Blue teams	Yes	Yes	OWASP
Nmap	Vulnerability scanning; Penetration testing; Red & Blue teams	Yes	Yes	CISA
OpenVAS	Vulnerability scanning; Penetration testing; Red & Blue teams	Yes	Yes	OWASP; CISA
sqlmap	Penetration testing; Red & Blue teams	Yes	Yes	OWASP; CISA
Metasploit	Penetration testing; Red & Blue teams	Yes (also has a commercial version)	Yes	OWASP; CISA
THC Hydra	Penetration testing; Red & Blue teams	Yes	Yes	OWASP
Snort	Red & Blue teams	Yes (commercial features)	Yes	NIST; CISA
Suricata	Red & Blue teams	Yes	Yes	CISA

6.1. Background on the GRACE Project at INOV

The GRACE project will be the focus of this case study. The DevSecOps framework developed in this study will be applied to its software development process, which enhances the DevOps methodology with principles of security-by-design and privacy-by-design.

INOV [5] is one of the 22 partners contributing to this project and has a team of three developers working on it. At INOV, the GRACE project is the responsibility of the cybersecurity research team. Because of this, the security and privacy principles previously described are familiar to the development team and considered by default. By also following DevOps methodology, the resulting software development process is closely aligned with DevSecOps, although not entirely identical. Specifically, the original DevOps pipeline consists of four distinct phases: Plan, Code, Build, and Release. The development architecture is based on microservices, enhancing flexibility and scalability. Each component is built using Jenkins, facilitating the automation of the building process, and then containerised using Docker, simplifying the packaging and delivery.

6.2. Applying Continuous Security Testing

Our DevSecOps framework includes three main constituents: a pipeline, a set of activities performed in each phase, and proposed tools to improve the performance and automation of these activities. Next, we compare these three components against GRACE's DevOps methodology to assess what changes and adaptations need to be made to apply the principle of continuous security testing.

6.2.1. Pipeline. The DevOps pipeline utilised in GRACE appears somewhat incomplete compared to the pipeline proposed in our study. Specifically, while our selected DevSecOps pipeline includes eight distinct phases (see Figure 1(d)), GRACE is structured around only four phases, lacking the Test phase and all three subsequent phases after Release, namely Deploy, Operate, and Monitor.

The DevOps pipeline for GRACE lacks a dedicated Test phase, primarily due to the project's testing activities being limited in scope. Currently, the project focuses solely on functional testing to verify if the application behaves as expected. However, a comprehensive testing approach, particularly including security testing, would be beneficial. The absence of Deploy, Operate, and Monitor phases is attributed to a different rationale. Being a research and innovation project, GRACE lacks a production environment, rendering the Deploy phase unfeasible. While the staging environment attempts to closely replicate a hypothetical production environment, its access is limited due to the collaborative nature of the project involving over 20 entities. Hence, integrating new or experimental security testing activities within this environment is not practicable.

Consequently, to transition GRACE's DevOps pipeline towards our proposed DevSecOps model, we integrated a new Test phase along with security testing activities into the SDLC at INOV. However, due to the previously mentioned constraints, the Deploy, Operate, and Monitor phases could not be implemented. This limitation unfortunately restricts our capacity to fully validate the effectiveness of the proposed DevSecOps framework.

6.2.2. Activities. To ensure the seamless integration of security testing activities into GRACE's SDLC, without causing delays or disrupting the workflow of developers, we aligned with the activities proposed in Table 3. Upon reviewing GRACE's pipeline, the following activities were considered suitable: test planning, test development, SAST, SCA, DAST, IAST, vulnerability scanning, and penetration testing. The red and blue teams activity was excluded due to the absence of an Operate phase in the project pipeline. Following the analysis presented subsequently, we established a finalised setup for continuous security testing activities in GRACE, as detailed in Table 5.

Beginning with the Plan phase, the only activity proposed for this phase is test planning. Since this activity is already being conducted in the project, there was no need to incorporate it into the SDLC anew. During the Code phase, the recommended activities include test development and SAST. Test development is another task already undertaken by the developers. In contrast, SAST represents a new addition, typically integrated through IDE plugins to assist developers in creating more secure code. The Build phase recommends two activities: SCA and SAST. Both are readily automatable using tools and can be seamlessly integrated into the development process.

In the newly added Test phase, three security testing activities were introduced: DAST, IAST, and vulnerability scanning. Both DAST and IAST are designed to be configured for fully automated execution. Vulnerability scanning, inherently automatic, has been incorporated into the SDLC. Accordingly, DAST has been adopted due to its automation capabilities. While IAST also offers potential for full automation, the absence of an updated open-source tool led to its exclusion from the GRACE project.

TABLE 5. ACTIVITIES PERFORMED IN GRACE: NEW ACTIVITIES IN **BOLD**, ACTIVITIES NOT PERFORMED IN ~~STRIKETHROUGH~~.

Plan	Code	Build	Test	Release
Test planning	Test development **SAST**	**SCA** **SAST**	**DAST** ~~IAST~~ **Vuln. scanning**	**Vuln. scanning** ~~Pen. testing~~

In the concluding phase of this pipeline, the Release phase, the security testing activities include vulnerability scanning and penetration testing. As previously mentioned, vulnerability scanning is designed for full automation, facilitating its straightforward implementation in this project. In contrast, penetration testing is inherently manual, relying on tools only to automate certain tasks. It requires specialised expertise that neither the developer team nor the author currently possess. As a result, penetration testing was not implemented in the GRACE project.

6.2.3. Tools. To determine the appropriate security tools for each activity, we initially referred to Table 4 as a foundation. Subsequently, a comprehensive discussion with the development team was conducted to precisely understand their specific needs. Given that GRACE is an active project, it was crucial to consider the programming languages and other tools already in use by the team. Predominantly, the project utilises Java and Python. Jenkins automates the build process and the creation of Docker containers, making the integration capabilities with Jenkins a priority. As for IDEs, Eclipse IDE, VS Code, and Sublime Text are the preferred ones.

Taking into account these factors, we selected the following tools: SonarQube Community Edition and SonarLint for SAST, Dependency-Check for SCA, and ZAP for both DAST and vulnerability scanning. With the exception of SonarLint, all these tools were integrated with Jenkins, enabling them to automatically run with each new build without requiring manual intervention. Next, we elaborate on the rationale behind this choice.

SAST is present in both the Code and Build phases. The SAST tools identified in Table 4 include SpotBugs, Clair, SonarQube, and PMD. Among these, SpotBugs functions solely as an IDE plugin, specific to the Eclipse IDE, and supports only Java. SonarQube stands out for its support of multiple languages and its capability to integrate with various platforms, including GitHub, Bitbucket, GitLab, and Jenkins. Therefore, we selected it for the Build phase. In the Code phase, while an IDE plugin solution is preferred, SpotBugs' limitations led us to consider SonarLint as a more suitable alternative. Its direct integration with SonarQube, support for a broad spectrum of languages, and availability for both Eclipse IDE and VS Code made SonarLint the optimal choice.

For SCA, we chose Dependency-Check. It supports several languages and file types. Its integration with Jenkins and SonarQube ensures compatibility with GRACE.

Next we have DAST activity, also with two potential tools: OWASP Zed Attack Proxy and Wapiti. Both of these tools are web scanners. OWASP ZAP has multiple ways of being used. It has a user-friendly GUI to be used

	SonarQube				
	General				
Project	Reliability	Security	Maintainability	Duplications (%)	Duplications
cleansing-reduction	E	A	A	8.60%	2030
data-retention-service	A	A	A	18.90%	119
dataset-ingestion	C	A	A	8.10%	434
docker-java	A	A	A	0.00%	0
docker-owncloud-client	A	A	A	0.00%	0
file-storage-api	E	A	A	0.00%	0
filepep	C	A	A	0.00%	0
files-deduplication	C	A	A	8.50%	1726
files-loader	C	A	A	11.10%	660
files-transformation	C	A	A	8.20%	400
graphdb-free-edition	A	A	A	0.00%	0
handling-metadata	A	A	A	20.50%	1655
import-export-service	A	A	A	10.70%	186
ingested-files-sharing	B	A	A	8.40%	571
metadata-catalogue	C	A	A	4.80%	920
ncmec-ingestion	C	A	A	7.10%	1345
oc-client-sync	E	A	A	0.00%	0
preprocessing-orchestration	A	A	A	0.00%	0
rdf4j-server	A	A	A	0.00%	0
scanning	A	A	A	8.20%	90
semantic-mpping	C	A	A	12.00%	3643
synthetic-referrals	E	A	A	9.20%	1816
workshop-support-page	C	D	A	9.00%	761
Average	C	A	A	6.67%	711.1

Figure 2. SonarQube overview results for the GRACE project.

manually or partially manually, but it also provides users with docker containers with predefined packages scans that are fully automatic. Wapiti also provides automatic scans through the use of the Command Line Interface (CLI). Although both scanners are suitable for DAST and can be used together with Jenkins, OWASP ZAP ended up being preferred due to its multiple modes of operation.

Lastly, for vulnerability scanning, given that OWASP ZAP is a web scanner that was also selected for the DAST activity, we also selected it for vulnerability scanning.

6.3. Results

The DevSecOps framework was then applied to the GRACE project, which follows a microservices architecture and it is composed of 23 different sub-projects. Each tool generated reports with the vulnerabilities found and their severity that will be analysed hereafter.

6.3.1. SonarQube results. SonarQube is a static code analyser that identifies a wide array of issues beyond security vulnerabilities, including code quality concerns, bugs, and code smells. The statistics produced for GRACE are summarised in Figure 2. This report assigns grades from A to E, with A being the best, across three key areas: code reliability (influenced by the number of bugs detected), security (based on the number of vulnerabilities found), and maintainability (which considers the effort required to address all code smells relative to the project's size). The presented data was collected by scanning each project on 18th August 2023.

The reliability of the sub-projects varies greatly, with some of them obtaining the highest grade (A),indicating minimal bugs, and others obtaining the lowest grade (E) because they have a lot of bugs or very serious bugs. Overall, the average Reliability is in the middle range (C). Security has the highest grade (A) for most projects, with the exception of *workshop-support-page* with a D grade. Regarding maintainability, all sub-projects achieved the maximum grade of A. Lastly, code duplication varies between projects, but most are below 10%.

SonarQube facilitates an Issue tab which shows all the issues detected according to three types, which are Bug, Vulnerability, and Code smell; and five levels of severity. SonarQube crates an additional table which is related to the Security Hotspots tab. Security Hotspots are pieces of code that need to be assessed in regards to security, as they might present a threat to the system. SonarQube classifies these security Hotspots based on review priority (High, Medium or Low) and attributed to a type. In GRACE, the majority of the sub-projects have 0 High priority security hotspots, the exceptions are *filepep*, *import-export-service* and *scanning* sub-projects which all have one high priority CSRF security hotspot. With Medium priority, the most common type is Permission (e.g. running images as root), and most sub-projects only have one Medium security hotspot. The Low priority cases present the most variability, both in number per project and in type. In general, the sub-projects with the most security hotspots are *handling-metadata*, *synthetic-referrals*, and *cleansing-reduction* which should be given more attention.

6.3.2. SonarLint results. SonarLint is tied to SonarQube, and the issues pointed out by both tools coincide. For this reason, we instead make an account of the development team's feedback regarding SonarLint's usefulness. A survey consisting of ten questions was sent to the three developers on the team after using the tool for one month.

The developers all had a positive experience with this tool and expressed their wish to continue using it for future projects. The installation and configuration of the tool was easy, although two developers reported some difficulties in installing and configuring it on the Eclipse IDE. Overall, they found SonarLint to be easy to use, useful, and very little intrusive. One developer had never used similar IDE plugins before and stated that SonarLint had a positive impact on their performance. The other two developers had already used other linting/SAST plugins before and felt that the impact of SonarLint on performance was neutral, but still pointed out that they found SonarLint more advantageous than the plugins they had used before.

6.3.3. OWASP Dependency-Check results. This SCA tool examines project dependencies to assess their security. It checks each dependency against vulnerability databases such as NVD, CISA Known Exploited Vulnerabilities Catalogue, Sonatype OSS Index, etc., and lists all the vulnerabilities found along with their severity level (Low, Medium, High, Critical). This tool was integrated with Jenkins and configured for all 23 sub-projects.

Figure 3, reveals a large number of detected vulnerabilities. The *semantic-mapping* sub-project stands out with a total of 322 vulnerabilities, followed closely by *preprocessing-orchestration* with 303 vulnerabilities. The *semantic-mapping* sub-project warrants particular attention due to harbouring the highest count of Critical vulnerabilities: 47 in total. A contributing factor to the large number of vulnerabilities (1795 across all projects) is the lack of updates to many dependencies since the beginning of the project over three years ago. Resolving these vulnerabilities varies in complexity; while many can be addressed by simply updating to the latest version of the affected library, others demand more substantial efforts.

6.3.4. OWASP Zed Attack Proxy results. OWASP ZAP is a web scanner used for DAST and vulnerability scanning. Therefore, this tool specialises in web applications

	Dependency-check (Severity)					
Project	Critical	High	Medium	Low	Total	Date
cleansing-reduction	5	17	29	1	52	08/08/2023
data-retention-service	0	0	0	0	0	08/08/2023
dataset-ingestion	6	17	29	1	53	08/08/2023
docker-java	0	0	0	0	0	08/08/2023
docker-owncloud-client	0	0	0	0	0	08/08/2023
file-storage-api	0	0	0	0	0	08/08/2023
filepep	0	0	0	0	0	08/08/2023
files-deduplication	6	17	29	1	53	08/08/2023
files-loader	6	17	29	1	53	08/08/2023
files-transformation	6	17	29	1	53	08/08/2023
graphdb-free-edition	29	42	97	11	179	08/08/2023
handling-metadata	6	12	24	1	43	08/08/2023
import-export-service	0	0	0	0	0	08/08/2023
ingested-files-sharing	6	17	29	1	53	08/08/2023
metadata-catalogue	27	63	64	7	161	08/08/2023
ncmec-ingestion	6	17	29	1	53	08/08/2023
oc-client-sync	0	0	0	0	0	08/08/2023
preprocessing-orchestration	26	147	108	22	303	08/08/2023
rdf4j-server	20	41	31	5	97	08/08/2023
scanning	0	0	0	0	0	08/08/2023
semantic-mpping	47	108	140	27	322	08/08/2023
synthetic-referrals	6	13	28	1	48	08/08/2023
workshop-support-page	4	12	223	33	272	08/08/2023
Total	**206**	**557**	**918**	**114**	**1795**	
Average	9.0	24.2	39.9	5.0	78.0	

Figure 3. OWASP Dependency-Check results for GRACE project.

	Zed Attack Proxy (Risk level)					
Project	Scan type	High	Medium	Low	Informational	Total	Date
file-storage-api	API scan	0	0	1	2	3	08/08/2023
handling-metadata	API scan	0	0	2	2	4	08/08/2023
workshop-support-page	Baseline scan	0	6	6	5	17	08/08/2023
Total		**0**	**6**	**9**	**9**	**24**	
Average		0.0	2.0	3.0	3.0	8.0	

Figure 4. OWASP Zed Attack Proxy results for GRACE project.

and APIs. Out of the 23 sub-projects that are part of the GRACE project, only 3 fit those categories, one of them is a regular web page, so the ZAP Baseline scan was used, and the other two are APIs, so the API scan was performed instead. The results can be seen in Table 4. ZAP generates a scan report listing the alerts found and categorises them into 4 risk levels, High, Medium, Low, or Informational. No High risk alerts were found, and the only Medium risk alerts (6) found belonged to the sub-project *workshop-support-page*. This sub-project had a total of 17 alerts, the most of the three projects, so it should be prioritised.

6.3.5. Critique. Although efforts were made to produce results as accurate as possible, there are some limitations that have a direct impact on the results. As mentioned previously, due to the research nature of the GRACE project, no production environment was present. In addition to that, the staging environment had strict access restrictions which made it impossible to apply the Deploy phase, Operate phase, and Monitor phase.

Similarly, not all proposed activities were performed. The activities from the absent phases were not executed, but even those related to the carried out phases were not fully performed either. Both IAST and Penetration testing are activities that were not fulfilled due to lack of tools and lack of expertise. Since it was not possible to fully apply the proposed DevSecOps framework, it was also not possible to fully assess the framework's validity.

Finally, another factor that heavily affects the results is the tools' limitations. Tools are not capable of detecting all defects (false negatives) and often report issues that are actually not a problem (false positives) [17]. The results presented in this study were not thoroughly analysed to remove false positives due to the lack of expertise of the author and the high volume of defects detected by the tools. For this reason, the amount of issues and defects detected by the tools is likely to be inflated and does not represent the real security status of the GRACE project.

7. Conclusions

In this paper, we focused on identifying a practical DevSecOps framework and in assessing it in a real-world environment. A CI/CD pipeline was defined based on existing literature, along with a list of activities to be performed in each phase. A list of tools was also created so that the activities described could be automated as much as possible. This framework was then applied to a case study. Although it was not possible to fully apply the DevSecOps framework due to the project's nature, our experiment was still a success. The developers found the tools useful and are willing to use them in future projects. Future work involves studying new metrics regarding the adoption and effectiveness of this framework, such as the perceived acceptance by the developers.

Acknowledgements: We thank the anonymous reviewers for their comments and insightful feedback. This work was supported by the Fundação para a Ciência e Tecnologia (FCT) under grant UIDB/50021/2020, by IAPMEI under grant C6632206063-00466847 (SmartRetail), and by the European Union's Horizon 2020 research and innovation programme under grant agreement No. 883341 (GRACE).

References

[1] 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*, 2022.

[2] F. Angermeir, M. Voggenreiter, F. Moyon, and D. Mendez, "Enterprise-Driven Open Source Software: A Case Study on Security Automation," in *ICSE-SEIP*, 2021.

[3] R. Brasoveanu, Y. Karabulut, and I. Pashchenko, "Security Maturity Self-Assessment Framework for Software Development Lifecycle," in *Proc. of ARES*, 2022.

[4] R. Desai and T. N. Nisha, "Best Practices for Ensuring Security in DevOps: A Case Study Approach," *Journal of Physics: Conference Series*, 2021.

[5] INOV, https://www.inov.pt/en/index.html, [n. d.], accessed: 2024-05-13.

[6] R. Kumar and R. Goyal, "Modeling continuous security: A conceptual model for automated DevSecOps using open-source software over cloud (ADOC)," *Computers & Security*, 2020.

[7] T. Lam and N. Chaillan, *DoD Enterprise DevSecOps Reference Design: Version 1.0*, 2019.

[8] X. Larrucea, A. Berreteaga, and I. Santamaria, "Dealing with Security in a Real DevOps Environment," in *Communications in Computer and Information Science*, 2019.

[9] T. Leppänen, A. Honkaranta, and A. Costin, "Trends for the DevOps Security. A Systematic Literature Review," in *Business Modeling and Software Design*, 2022.

[10] R. Mao, H. Zhang, Q. Dai, H. Huang, G. Rong, H. Shen, L. Chen, and K. Lu, "Preliminary Findings about DevSecOps from Grey Literature," in *QRS*, 2020.

[11] V. Mohan, L. ben Othmane, and A. Kres, "BP: Security Concerns and Best Practices for Automation of Software Deployment Processes: An Industrial Case Study," in *SecDev*, 2018.

[12] F. Moyón, R. Soares, M. Pinto-Albuquerque, D. Mendez, and K. Beckers, "Integration of Security Standards in DevOps Pipelines: An Industry Case Study," in *Proc. of PROFES*, 2020.

[13] H. Myrbakken and R. Colomo-Palacios, "DevSecOps: A Multivocal Literature Review," in *Software Process Improvement and Capability Determination*, 2017.

[14] T. Rangnau, R. V. Buijtenen, F. Fransen, and F. Turkmen, "Continuous Security Testing: A Case Study on Integrating Dynamic Security Testing Tools in CI/CD Pipelines," in *EDOC*, 2020.

[15] M. Sánchez-Gordón and R. Colomo-Palacios, "Security as culture: A systematic literature review of devsecops," in *Proc. of ICSEW'20*, 2020.

[16] A. Sojan, R. Rajan, and P. Kuvaja, "Monitoring solution for cloud-native DevSecOps," in *SmartCloud*, 2021.

[17] N. Tomas, J. Li, and H. Huang, "An Empirical Study on Culture, Automation, Measurement, and Sharing of DevSecOps," in *Cyber Security*, 2019.