FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Understanding Developer Reasoning in Addressing Vulnerability Alerts

Inês Sá Pereira Estêvão Gaspar

Mestrado em Engenharia Informática e Computação

Supervisor: Professor Doutor Rui Maranhão

Co-Supervisor: Sofia Oliveira Reis

July 24, 2025

Understanding Developer Reasoning in Addressing Vulnerability Alerts

Inês Sá Pereira Estêvão Gaspar

Mestrado em Engenharia Informática e Computação

July 24, 2025

Resumo

Os analisadores estáticos são ferramentas essenciais para ajudar os desenvolvedores a identificar e reduzir vulnerabilidades nas fases iniciais do ciclo de desenvolvimento de *software*. Estas ferramentas analisam automaticamente o código e produzem uma lista de alertas para potenciais problemas de segurança. Contudo, os desenvolvedores frequentemente reportam falhas, particularmente em relação à clareza e utilidade desses alertas. Muitas vezes, as mensagens não têm contexto, contém termos técnicos e não explicam claramente a vulnerabilidade, o que causa confusão, frustração e, em muitos casos, leva ao abandono destas ferramentas. Apesar do uso generalizado dos analisadores estáticos, pouco se sabe sobre a forma como os desenvolvedores interagem com os alertas ou sobre o tipo de apoio de que necessitam durante o processo de resolução. Esta falta de compreensão dificulta o *design* de ferramentas que estejam alinhadas com os fluxos de trabalho reais e as exigências cognitivas dos desenvolvedores.

Tendo como objetivo colmatar esta lacuna e capacitar um melhor *design* das ferramentas, esta tese explora como os desenvolvedores interpretam e respondem a alertas produzidos por analisadores estáticos de código. Assim, investiga os processos de raciocínio (cadeia de pensamento) dos desenvolvedores, as suas reações e comportamentos na tomada de decisões, bem como os desafios que enfrentam ao tentarem compreender e agir perante alertas de segurança. Para isso, foi realizado um *user study* com 12 participantes com diferentes *backgrounds*, níveis conhecimentos em segurança e experiência com ferramentas de análise estática.

O estudo demonstra que, em geral, os desenvolvedores quando se deparam com este género de tarefa seguem uma cadeia de pensamento composta por cinco fases: (1) compreender o alerta, (2) analisar o contexto do código, (3) determinar a ação a tomar, (4) implementar o *fix* e (5) testar e validar a solução. Os participantes consideraram esta tarefa moderamente exigente, mas não particularmente stressante. No entanto, surgiram dificuldades significativas ao interpretar termos desconhecidos (na Fase 1) e ao tentar compreender a vulnerabilidade sem exemplos ilustrativos (na Fase 4). Para ultrapassar estes desafios, os participantes recorreram a recursos externos como o Google, o ChatGPT, o GitHub ou o StackOverflow, para encontrarem explicações, exemplos ou casos semelhantes.

Os resultados sugerem que os desenvolvedores seguem uma cadeia de pensamento consistente ao lidar com alertas de segurança e expõem lacunas críticas no suporte fornecido pelas ferramentas de análise estática atuais. Os participantes tiveram que utilizar frequentemente recursos externos como o ChatGPT, o Google para colmatar a falta de clareza ou de explicações nos alertas. Isto mostra que alinhar o *output* dos analisadores estáticos de código com os processos de raciocínio dos desenvolvedores - através de mensagens claras e orientação contextual - poderá melhorar significativamente a usabilidade das ferramentas, promover a confiança e apoiar práticas de programação segura mais eficazes.

Palavras-chave: análise estática, compreensão de alertas, vulnerabilidades, cadeia de pensamento

Abstract

Static analysis tools are essential for helping developers identify and reduce software vulnerabilities early in the development life cycle. These tools automatically scan source code and produce alerts for potential security issues. However, developers frequently report shortcomings, particularly with the clarity and usefulness of these alerts. Messages often lack context, contain technical jargon, and fail to explain the vulnerability clearly, which leads to confusion, frustration, and in many cases, abandonment of the tool. Despite the widespread use of static analyzers, little is known about how developers engage with these alerts or what support they need during the resolution process. This lack of understanding makes it difficult to design tools that align with developers' real-world workflows and cognitive demands.

Aiming to address this gap and inform better tool design, this thesis explores how developers interpret and respond to alerts produced by static analysis tools. It investigates their reasoning processes (chain-of-thought), reactions, and decision-making behaviors, as well as the challenges they face when attempting to understand and act on security-related alerts. To this end, a user study was conducted with 12 participants from diverse backgrounds, with varying degrees of security knowledge and experience with static analysis tools. The study reveals a recurring five-phase chain of thought: (1) understanding the alert, (2) analyzing the surrounding code context, (3) deciding on a course of action, (4) implementing the fix, and (5) testing and validating the solution. Participants generally found these tasks moderately demanding but not overly stressful. However, significant difficulties arose when interpreting unfamiliar terms (during Phase 1) and understanding the vulnerability without concrete examples (during Phase 4). To overcome these challenges, participants turned to external resources, such as Google, ChatGPT, GitHub, or Stack Overflow, to search for clarifications, examples, or similar cases.

These findings suggest that developers follow a consistent chain of thought when addressing security alerts and expose critical gaps in current tool support. Participants frequently turned to external resources to compensate for unclear or under-explained alerts. This suggests that aligning static analysis output with developers' reasoning processes, through clear messages and contextual guidance, could significantly improve tool usability, foster trust and support more effective secure coding practices.

Keywords: static analysis, alerts comprehension, vulnerabilities, chain of thought

UN Sustainable Development Goals

The United Nations Sustainable Development Goals (SDGs) are a global guide composed by goals and target actions in order to transform and build a better and more sustainable future. This guide includes 17 goals to address the biggest challenges in the world. [Nations, 2025]

When analyzing the objectives of this thesis, as well as their implications and future applications in light of the SDGs, its contributions can be included in the following goals:

SDG 4 Ensure inclusive and equitable quality education and promote lifelong learning opportunities for all

SDG 8 Promote sustained, inclusive and sustainable economic growth, full and productive employment and decent work for all

SDG 9 Build resilient infrastructure, promote inclusive and sustainable industrialization and foster innovation

SGD	Target	Contribution	Performance Indicators and Metrics
4	4.4	Facilitate the interpretation and learning of technical concepts and problems, as well as good practices to be adopted in software development by improving the messages of the Static Analysis tool alerts, making them clearer and more informative.	Level of understanding of alerts. Feedback from developers on alert messages.
8	8.2	Improving static analyzers can increase developer productivity and reduce the time spent interpreting alerts and resolving them, which contributes to reducing stress and improving development flow.	Level of frustration. Time taken to resolve problems detected.
9	9.5	Improving static analysis tools, in order to understand how they can respond to the needs of the developers who use them, making them more robust, efficient systems and increasing their adoption.	Level of frustration. Developer feedback. Alert resolution time.

Acknowledgements

First of all, I would like to express my sincere gratitude to Professor Rui Maranhão and Professor Sofia Reis for their close guidance and for all their unwavering support during the completion of this Master's thesis.

I am also deeply thankful to researcher Cláudia Mamede for helping to develop the study and reviewing the document. Her experience and feedback were essential for the study design, as was her support in recruiting participants. I extend this thanks to researcher Leonardo Fernandes for his helpful advice during the writing phase and his review of the final document.

I would like to thank all the participants in the study for their availability, engagement, and interest at every stage. Their contribution was fundamental to the success of this work.

I am grateful to my friends for all their help and all the moments we shared along the way.

Finally, I wish to express my heartfelt thanks to my family for their endless patience, support, and kindness during all these years.

Inês Gaspar

"Sempre chegamos ao sítio aonde nos esperam."

José Saramago

Contents

List of Figures

List of Tables

List of Acronyms

AI	Artificial Intelligence
CoT	Chain of Thought
CVE	Common Vulnerabilities and Exposures
CWE	Common Weakness Enumeration
DOM	Document Object Model
IDE	Integrated Development Environment
JS	JavaScript
LLM	Large Language Model
SA	Static Analysis
SAST	Static Application Security Testing
OSV	Open Source Vulnerabilities
XSS	Cross-Site Scripting

Chapter 1

Introduction

Static analysis (SA) tools play a central role in modern software development. By examining source code without executing it [Li et al., 2025], static analyzers identify bugs, code smells, and security vulnerabilities such as memory leaks, null pointer dereferences, and injection flaws [Li et al., 2024, Livshits and Lam, 2005]. Their ability to surface issues early in the software development process has made them widely adopted in the industry [Do et al., 2022].

However, the practical effectiveness of SA depends not only on the precision of the analysis itself (i.e., how accurately and meaningfully it detects issues) but also on how developers interpret and respond to the alerts it generates. Prior work has shown that these alerts are often overly technical, cryptic, and difficult to act upon [AlOmar and Mkaouer, 2024, Yang et al., 2020, Nachtigall et al., 2019]. This lack of clarity can hinder developers, especially those with less experience or limited security expertise, from understanding the reported issues [Braz and Bacchelli, 2022, Tooley, 2024]. As a result, they may overlook critical vulnerabilities, apply incorrect fixes, or become discouraged by the volume of false positives and low-priority warnings [Nachtigall et al., 2022, Alahmadi et al., 2022]. In response to these challenges, research has advanced primarily in two directions. One line of work focuses on improving the underlying analysis itself – by increasing detection accuracy, reducing false positives, or scaling analysis to large codebases [Venkatesh et al., 2024, Mohajer et al., 2024, Christakis and Bird, 2016]. Another line focuses on improving the way analysis results are presented to developers, through better alert prioritization, visualization, or integration into development environments [Mao et al., 2025, Mamede, 2025].

Despite these efforts, little is known about how developers reason through static analysis alerts in practice and the cognitive strategies they employ to interpret them. Concretely, the strategies they employ to understand alerts, locate relevant code, and reason through possible fixes remain poorly understood. These behavioral dimensions are essential to the practical effectiveness of static analysis, yet have received limited empirical attention.

1.1 Motivation

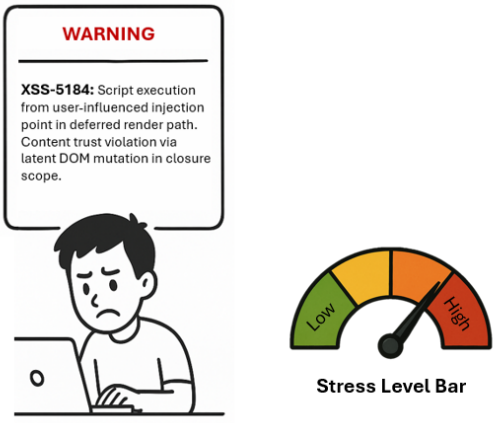

Figure 1.1: Poor and confusing warning messages (image created with ChatGPT)

Figure 1.1 presents an example of a security alert that a programmer might face while coding: “XSS-5184: Script execution from a user-influenced injection point in the deferred render path. Content trust violation via latent DOM mutation in closure scope".

The depiction of a confused expression alongside an elevated stress indicator reflects a common scenario in software development: a highly technical, vague, or abstract security warning, which provides little to no guidance regarding its cause or fix. Although such alerts are critical for protecting software and applications, they are often formulated in a manner that fails to communicate effectively with the developers responsible for addressing them. This often leads to frustration, stress, and, in some cases, dismissal of warnings without proper remediation, jeopardizing the security and quality of the code [Johnson et al., 2013].

This problem is particularly concerning given that even experienced developers often struggle with difficulties when attempting to use static analyzers effectively. These challenges are due to an overwhelming volume of unclear alerts, poor integration into development workflows, and a lack of actionable guidance. Consequently, these tools are frequently underused, and their alerts disregarded, allowing security issues to reach production and enabling the persistence of bad practices [Nachtigall et al., 2019].

The motivation for this thesis is to understand the main bottlenecks and chain of thoughts followed by developers when interpreting and resolving warnings produced by a static analysis tool. These conclusions can be used in the future as ways to customize and improve step-by-step reasoning solutions that could complement the reports that currently exist.

Therefore, this project is based on two major problems related to static analysis that were intended to explore and investigate:

- **Security Knowledge Gap.** Static analysis tools often generate highly technical and complex outputs that developers, especially those without extensive security expertise, find difficult to interpret. This knowledge gap between the tool’s output and the developer’s under-

standing leads to missed vulnerabilities, incorrect fixes, and a lack of clarity on why specific issues matter. Developers struggle to translate cryptic alerts into actionable fixes due to limited security training and the complexity of the code issues identified.

- **Poor Understanding of Secure Coding Practices.** Beyond fixing immediate issues, developers must improve their understanding of secure coding practices and code quality best practices. Current static analyzers do not provide sufficient guidance for developers to learn from their mistakes and avoid them in the future. Many developers lack awareness of common vulnerabilities, which hampers their ability to proactively write secure, high-quality code.

1.2 Research Objectives

This thesis investigates how developers behave when responding to SA alerts. Rather than proposing new tooling or refining alert content, our focus is on observing and characterizing the reasoning processes that developers follow as they interpret and act on warnings. By identifying common strategies, points of confusion, and decision-making patterns, this work aims to surface behavioral insights that can inform future assistive mechanisms. In particular, these insights may support the integration of step-by-step reasoning into Large Language Models (LLMs), enabling them to serve not only as code assistants but also as educational tools that reflect and reinforce how developers naturally approach security-related issues. Hence, we defined the following objectives:

- **Collect the chain of thought inherent in understanding and solving warnings produced by static analyzers:** to understand what challenges and difficulties junior developers face with static analysis tools, behavioral patterns and lines of reasoning, and to analyze the type of needs and information that are not present in the reports they produce.
- **Capture emotional and cognitive factors**: observe and collect factors that shape the interpretation and fixing the alerts, such as:
  - Cognitive Load and Frustrations: signs of confusion, stress, or overwhelm that indicate unclear or complex alert that require excessive mental effort.
  - Confidence and Trust: feeling like confidence, assurance or uncertainty reveals the degree to which developers trust the alert and self-efficacy in resolving security issues.
  - Emotional Engagement and Framing: reflects how developers perceive the alert (usefulness versus noise) and whether it is actionable or not, which evaluates developer engagement and motivation to act, a central piece to the adoption of static analysis tools.
- **Categorizing bottlenecks and chain of thought:** categorize bottlenecks and challenges experienced when developers try to understand the alerts produced by SA tools, as well as

the chain of thought followed, in order to be possible to create solutions based on these needs in the future.

By studying how developers reason through SA outputs, this thesis provides insights that can inform the design of more effective and educational tool support, aligned with real developer needs and reasoning practices.

1.3 Research Questions

This dissertation aims to answer two research questions, one on behavioral patterns and the other on cognitive challenges:

RQ1: ***How do developers respond to static analysis reports for security vulnerabilities?***

This question focuses on understanding and analyzing how developers perceive, interpret and react to security-related alerts. The goal is to identify common behavioral patterns and decision-making processes, including emotional and contextual factors influencing whether an alert is addressed or ignored. Thus, the aim is to go beyond surface-level actions by uncovering the underlying motivations, such as a lack of understanding, perceived irrelevance, or external pressures like time constraints. Analyzing these factors makes it possible to improve the design of tools that better align with developer needs and encourage more effective responses to vulnerabilities.

RQ2: ***What questions do developers ask, and what challenges do they face?***

This question explores the cognitive difficulties developers encounter when interpreting static analysis reports. By observing where developers get stuck, the help they request and documenting the questions they ask, the goal is to identify key pain points in understanding and acting on tool feedback. Uncovering what information is missing or unclear can guide the design of more supportive, developer-centric tools, ultimately increasing adoption and effectiveness.

1.4 Dissertation Structure

In addition to the Introduction, which gives context to the topic of the dissertation, as well as the motivation for its exploration and objectives, this document contains five more chapters.

In the second chapter, State Of The Art, the state of the art is described and related work is presented. They mentioned the fields that are also explored directly or indirectly in this project. It begins with the Background and is followed by Static Analysis Limitations and Bottlenecks and then some research developments in the areas of LLMs in Static Analysis, Code Understanding, and Improving Alerts Messages.

The third chapter, Vulnerability Alerts: Understanding Developer Reasoning, details the study conducted to accomplish the objectives of this thesis. It outlines the Methodology, in which each phase is described. And finally, it presents the profiles of the participants involved in the study in Participant Background and Profiles.

In the fourth chapter, Results and Discussion, the results of the study are presented and analyzed in relation to the research questions in Behavioral Patterns (RQ1) and Cognitive Challenges (RQ2). It also includes a Discussion of the findings and ends with Threats to validity.

The fifth chapter, Developer Reasoning Improves LLM Explanations introduces the design of a proof-of-concept solution in Prompt Variants, is followed by the presentation of the Reasoning-Aligned Assessment and finished with a discussion of its implications in Discussion and Takeaways.

Finally, in the sixth chapter, Conclusion and Future Work summarizes the main findings and contributions of this work and the future work that can be applied with this knowledge.

Chapter 2

State Of The Art

This chapter presents an exploration of research related to static analysis, innovations in the area, namely integrations with LLMs, and studies related to the application of LLMs in the context of code understanding.

It begins with a brief contextualisation of the general panorama of the area of static code analysis, followed by a description of the main gaps and challenges experienced by developers when using SA tools and, finally, an analysis of the most recent articles that reflect the advances made in recent years with the introduction of LLMs to improve currently existing SA solutions and the way they help with code understanding tasks.

2.1 Background

Static analysis is a software methodology that examines code (source code) without executing it. Its purpose is to detect and report issues related to security and best practices. These reports are triggered when suspicious patterns or vulnerabilities are identified in code. This technique allows the detection of issues early in the development cycle. It is included in Code Review as part of the Implementation phase of the Security Development Lifecycle (SDL) [OWASP, 2025b]. As a result, static analysis is often integrated into development environments (IDEs) and continuous integration pipelines (CI / CD), enabling problems to be identified and addressed before reaching production [DataDog, 2025].

Despite their advantages, static analysis tools face significant limitations in the quality of the results generated. The outputs are generally described as vague, overly technical, or filled with jargon. In many cases, they produce an overwhelming number of warnings, which makes it difficult for developers to act on them. So, developers find themselves struggling to understand the alert, assess its relevance, or determine how to fix it. The situation is further exacerbated by the high rates of false positives, alerts that erroneously flag the code as dangerous. This fact can diminish developers' trust in the tools and also lead to warnings being ignored. Finally, another key lies in the prioritization of alerts. Due to the lack of clarity and contextual information in the messages, developers find it hard to distinguish between critical issues and less important ones.

Enhancing the informativeness and relevance of alerts is crucial for increasing tool adoption and the quality of the software being developed.

In recent years, with the increase in the use of LLMs and the quality of their results, there has been more research into how LLMs can be integrated into software development tools. This research has been carried out at the various stages of software development due to their capabilities in software construction tasks such as code generation, testing, debugging and code review. Studies have been carried out on the intersection of this technology with static code analysis tools and how these can be improved (e.g. weakness detection, performance, patch generation, prioritization and filtering of alerts, etc.). In the research carried out, the emphasis has been on this aspect, as well as on the reported limitations of static analysis tools and the causes that lead to their abandonment. However, little attention has been paid to the quality and resolution of the complexity inherent in alerts.

The research and exploration of this field performed to date has centred on the general limitations and bottlenecks of these static analysis tools (mainly in terms of operation), on analyzing the interaction of developers with LLMs, particularly in code understanding, code review and debugging.

This is followed by an overview of the most recent studies on static analysis field and the application of LLMs in these tools, as well as in the context of code understanding.

2.2 Static Analysis Limitations and Bottlenecks

There are several studies aimed at investigating the reasons why developers abandon and are dissatisfied with the use of static analysis tools. It is important to first identify the limitations of these tools to improve the design of better and more suitable solutions. In order to gain a better understanding of the main concerns and difficulties experienced by developers, Table 2.1 shows data collected from research in the area.

Criteria	**Reason**	**Consequences**	**Article**
False positives	Analysts spend a lot of time validating alarms manually. Require monotonous tasks to reduce false positives.	Alarm desensitization. Mistrust in the tools. Lack of resolution of alarms. No prioritization of alarms.	[Alahmadi et al., 2022] [Johnson et al., 2013] [Nachtigall et al., 2019] [Nachtigall et al., 2022]
Lack of context in alarms	-	Confusion on tool adequacy. Ignore the alarms.	[Alahmadi et al., 2022]
Loosely written signatures	To capture a wide range of activities and all the possible behaviors.	Signature is not useful. Alarms cannot be reviewed.	[Alahmadi et al., 2022]
Bad description on the alarm and lack of information	Incomprehensible and short descriptions. No explanation of the warning/vulnerability or why the warning was triggered. No documentation about the issue detected and what might be wrong with the code.	Need to research and obtain more information. Spend time validating the alarm, reducing productivity. Alarm desensitization.	[Alahmadi et al., 2022] [Johnson et al., 2013] [Nachtigall et al., 2019] [Nachtigall et al., 2022]

Poorly presented output	Not user-friendly, nor intuitive No structure.	Frustration. Waste a lot of time trying to figure out what needs to be done.	[Johnson et al., 2013]
Lack of support	Difficulty in sharing the setting with the team. Manual process and confusion when the standards changed.	Demotivate developers.	[Alahmadi et al., 2022]
Lack of or ineffectively implemented fixes	Difficulty in offering quick fixes, especially for more complex issues. Sometimes, ineffective fixes are provided.	Difficulty in understanding the problem and the fix. Reduce productivity. Wrong fixes. Diminish trust in the tools.	[Johnson et al., 2013] [Nachtigall et al., 2019] [Nachtigall et al., 2022]
Workflow Integration	Some tools are separate from the environments that developers use	Configuration challenges. Difficulty releasing when the analysis should be executed.	[Johnson et al., 2013] [Nachtigall et al., 2019]
Lack of customization	No customization of analysis rules. Make it easier to select rules and not have them all active by default.	Increases the number of alerts, some of which are irrelevant depending on the context.	[Nachtigall et al., 2022]

Table 2.1: Factors that cause dissatisfaction with SA tools

From the limitations listed in Table 2.1, this thesis will explore the absence of alert context and vague, hard-to-understand messages. The next section will cover studies related to these two areas, as well as other improvements that indirectly affect message quality, such as advancements in bug detection.

2.3 Related Work

The following sections explore and publicize some of the innovations recently studied in the field of static analysis, especially with the integration of LLMs. Given the few studies published in the area of understanding developers' difficulties in analyzing and interpreting alerts, some of the sections presented focus on some of the shortcomings pointed out in the previous table, as well as the use of LLMs for Code Understanding.

2.3.1 LLMs in Static Analysis

One project created by [Mohajer et al., 2023] involved developing SkipAnalyzer, a tool that leverages Large Language Models for static code analysis. This proof of concept was structured around three core functions: 1) detecting and reporting errors, 2) filtering out false positives from the results, and 3) correcting detected errors. The aim was to demonstrate the capabilities of LLMs, specifically GPT, in these tasks. SkipAnalyzer was evaluated on two critical error types: Null Dereference and Resource Leak. The study demonstrated that SkipAnalyzer excels in certain static analysis tasks. Specifically, for error detection, it achieved an accuracy of 68.17% for Null Dereference and 76.95% for Resource Leak. This represents an improvement over the tools currently used (compared to Infer), with accuracy increases of 12.86% and 43.13%, respectively. Regarding reducing false positives, SkipAnalyzer reached accuracies of 93.88% for Null Dereference and 63.33% for Resource Leak. Additionally, in error correction, SkipAnalyzer successfully generated syntactically correct fixes for the detected errors.

Recent studies have explored the integration of Large Language Models (LLMs) like ChatGPT with static analysis tools such as PMD to enhance software quality education. This paper presents an experimental study where undergraduate and graduate students used ChatGPT alongside PMD to address code quality issues. The study's notable findings include identifying specific PMD rule categories—such as Design, Documentation, and Security—that are most commonly addressed by students. Interestingly, the research revealed that students could engage more effectively with the debugging process when assisted by ChatGPT, which provided explanations and potential fixes for the identified issues. This collaborative use of LLMs and static analyzers not only improved the students' understanding of code quality issues but also helped in cultivating a practical skillset in debugging and software maintenance [AlOmar and Mkaouer, 2024]. Such educational integration of LLMs with static analysis tools can significantly enhance learning experiences in software engineering education. By providing real-time, context-aware feedback and explanations, LLMs like ChatGPT can help demystify the often cryptic warnings generated by static analyzers, making them more accessible and educational. This approach can especially be beneficial in teaching secure coding practices, as it allows students to interactively learn and understand security vulnerabilities and their fixes, thereby preparing them to handle real-world security challenges more effectively.

The research of [Hao et al., 2023] focused on understanding how LLMs can improve static analysis tasks. In this sense, this study developed an approach, E&V (Execution&Verification), which used LLMs to simulate the execution of pseudocode, without human supervision and without the need for external verification. The fact that there is verification of the pseudocode reduces the risk of hallucinations associated with LLMs and, in turn, increases the accuracy of the results. Furthermore, since LLMs are not limited to understanding specific programming languages, they can perform static analysis for pseudocode in different languages. This prototype, together with GPT4, was applied to triage crashes for the Linux kernel. The evaluation of this prototype was carried out on 7 categories of bugs, corresponding to 170 recent crashes. It was concluded that this

tool was able to identify the blamed function in 81% of cases. It was also found that the execution verification process increased the accuracy of the implemented tool from 28.24% to 81.18%. Thus, E&V presented promising results that allow for a reduction in human effort and the level of expertise in static analysis tasks.

The studies analyzed in this section focused on trying to improve SA tools in general, for example, by making them more complete with error detection, false positive filtering and correction of the errors found.

2.3.2 Code Understanding

Code comprehension is an important field in Software Development, which refers to the ability to understand the structure, interpret and analyze code. This skill is essential for tasks such as maintenance, bug detection and correction, code review and, consequently, for interpreting alerts produced by SA tools, in order to be able to identify the causes of their being triggered. Thus, studying recent advances in code understanding helps to better understand the challenges in software analysis and explore approaches that can make the interpretation of alerts more effective.

The work of [Sheese et al., 2024] focuses on understanding the patterns of help requested by students from LLM tools. This study was carried out in an academic environment on a 12-week introductory programming course, where students had to ask their technical questions first to the tool developed for this purpose (CodeHelp) in order to collect all the queries made for later analysis. The tool used was based on LLMs (GTP 3.5 turbo), but with the particularity of not directly revealing the solution to the reported problems, if they were practical questions. In this way, it was concluded that most of the queries immediately asked for help with programming assignments, while few were related to understanding and deepening concepts. It was also possible to categorize the queries into three large groups: Debugging, Implementation and Understanding. In terms of results, it was found that 48% of the queries were related to debugging, 38% were related to implementing exercises and 9% were related to understanding, with the remainder being unrelated or out of context queries. Finally, it was found that the students provided the tool with the minimum information in their queries (which in this case asked them to fill in several fields: Language, Code, Error Message and Question), which proves that this is an area that they benefited from learning.

Another research done by [Roest et al., 2024] was carried out in an educational context in order to understand the contribution that LLMs can make to programming education through the automation of next-step hints and feedback for students in the context of solving programming problems. Initially, they began by analyzing and understanding what kind of prompts could generate quality and satisfactory next-step hints. At this stage, it was concluded that the best prompts only included a brief description of the problem and specific keywords, such as ‘hint’ and ‘student’. With this information, a tool was built to support students in this area, StAP-tutor, with the aim of enabling learning rather than providing a solution to the problem. It was evaluated through experiments in which students had to solve certain programming problems in order to extract results on the type of hints and their quality. In this sense, it was found that the tool most often

provided specific feedback that described the next-step hint in a personalised way that was appropriate to the code and approach that the student had implemented, as well as an explanation for the hint provided. However, sometimes the hints also contained wrong or misleading information, and it did not provide support for all the problem-solving phases, particularly at the end of the assessment.

[Haonan Li, 2023] employed an investigation to study where and how LLMs can help static analyzers by asking appropriate questions. This study used a specific error-finding tool, UBITect, producing many false positives from static analysis. The development of this study was based on understanding the summarization of functions since this element plays a significant role in the chosen tool, as it has a high false positive rate due to its inability to recognize special functions and not recognizing postconditions. This research concludes that false positives could be effectively eliminated if the questions were carefully constructed. In this sense, it was needed to pay attention to the characteristics of LLMs (such as the tendency to hallucinate and provide unreliable answers) and follow some fundamental principles of prompt design, namely Chain-of-Thought, Task Decomposition and Progressive Prompt), considering the behaviors or summaries of the functions.

As mentioned at the beginning of the section, there are also some studies on how this integration can be applied and what influence it has in the educational field, although they are limited. Thus, a study was carried out to find out what kind of help students asked LLMs to understand the practical content and implementations requested (insert quote), and another to find out how students used these two technologies in a complementary way to understand the warnings coming from the SA and for code review.

These studies focus on understanding how developers interact with LLMs during the various phases of software development, mainly in the generation of code and in understanding the bugs found. Thus, these studies are based on experiences with the implementation of tasks using and supporting LLMs. This is exactly the focus of this dissertation, but in a different component, since it aims to understand what kind of questions developers ask LLMs when they find warnings produced by SA that they don't understand, and how this information can be included in static analysis tools.

2.3.3 Improving Alerts Messages

In recent years, there has been greater concern about making the alerts produced by SA tools clearer and more informative. The messages they contain are often characterized as too general, technical (jargon), and lacking in context, which hinders developers' interpretation and contributes to their being devalued due to their intelligibility. Progress has been made to enrich these messages with additional information, such as better explanations of the cause of the problem, ways of exploiting the faults, and suggested fixes for the issue detected. Many of the strategies studied to work on this problem use LLMs to generate descriptions based on the original alert message, adapted to the specific context of the code. These improvements make alerts easier to understand, actionable, and contribute to greater effectiveness of static analysis tools in the software development process.

[Chapman et al., 2024] proposed enhancing static program analysis by integrating LLMs to improve error specification inference, particularly for C programs. This method interleaves static analyzer queries with LLM prompts, where intermediate static analysis results dynamically influence the LLM's input and vice versa, making the analysis more context-aware and accurate. The study revealed significant improvements in both recall and F1-score when comparing the enhanced approach against the traditional static analysis. For example, recall increased from an average of 52.55% to 77.83%, and F1-score improved from 0.612 to 0.804. These results indicate that the LLMs were particularly effective in interpreting complex error-handling patterns in code, thereby reducing the instances of overlooked or misinterpreted error specifications. By providing explanations and contextual insights generated by LLMs, users can gain a deeper understanding of security-related code anomalies and their implications. This approach could be further developed to tailor developers' needs when they have to resolve these alerts, improving your error messages and equipping programmers with secure coding practices.

While the work of [Mamede, 2025] focuses on the interpretation and comprehension of alerts by improving alert messages. This project created a tool, SECGen, which parses alerts produced by static analyzers and restructures them so that they are more structured, clear and actionable for all developers, not just experts. They then validated this solution in a user study, where they compared the interpretability of the alert produced by SECGen with that of alerts produced by other static analyzers currently in use, in this case, CodeQL. The conclusions were that developers who used interpretable reports (produced by SECGen) understood and corrected the reported vulnerabilities better, being 33% faster than those who used tools like CodeQL, and it was found that they produced more correct patches. In addition, the feedback was that the clear and modular structure of the alert was critical to its effectiveness, stating that these aspects are just as important as the technical context present.

While the investigations discussed herein have provided valuable contributions toward improving the quality of messages in SA tool reports, their methodological and conceptual scopes differ from the current undertaking. Specifically, the extant research has largely concentrated on the construction of complementary solutions to enhance the informativeness and usability of warning messages. Conversely, the objective of this project is to comprehend the intricate chain of reasoning employed by developers and to ascertain the key difficulties they encounter when confronted with SA reports. This foundational understanding is crucial for subsequently developing targeted solutions that effectively address developers' demonstrated needs.

Chapter 3

Vulnerability Alerts: Understanding Developer Reasoning

This chapter presents the user study designed to analyze how developers respond and react to reports produced by static analysis tools. It describes the methodology and protocols created, the profile of the participants, and the challenges faced during this process.

3.1 Approach

Based on the analysis in the previous chapter, it is clear that while some studies have examined how developers perceived security warnings, several important gaps remain. There is a lack of research focusing on developers with limited security expertise, especially regarding their initial reactions, reasoning processes, and how they compensate for knowledge gaps when interpreting alerts, including the use of external resources. This study was designed to investigate how developers perceive and experience these flaws and to identify the key challenges they face when interacting with static analysis reports for security issues.

These objectives were accomplished by using the following strategies:

- **Understanding Developers' Mental Models:** Employ qualitative methods—such as think-aloud protocols, interviews, surveys—to capture the questions developers ask, strategies they use when addressing security alerts.
- **Identifying Gaps and Challenges:** Analyze recurring difficulties in interpreting alerts and applying fixes, to uncover unmet needs and areas for improvement in static analysis tools.

3.2 Methodology

This section outlines the study design and methodology, detailing each of its three main phases: the **Pre-Screening**, **Hands-On Experiment** and **Post-Interview Survey**. It also describes the data

protection protocols and precautions taken to ensure participant privacy and ethical compliance. The study was structured into three distinct phases:

- **Pre-Screening Survey:** Collected data on participants' backgrounds, experience levels, behavioral tendencies, and initial reactions to alerts.
- **Hands-On Experiment:** Observed participants as they interacted with real security reports, capturing their thought processes and problem-solving strategies.
- **Post-Experiment Survey:** Gather insights into participants' experience, decision-making process, and the types of external resources they relied on—including AI tools—when interpreting and resolving the alert.

Before detailing each phase, it is important to describe how the environment was set up and which tools and technologies were used to support the study.

3.2.1 Study Environment and Tooling Setup

A dedicated development environment was created for the study to ensure participant privacy and simplify setup. This environment was hosted in GitHub Codespaces using a specially created account that could be safely shared with participants. The setup included the original repository containing the vulnerable code and the CodeQL static analysis tool, enabling a realistic simulation of a security-focused development workflow. It was important that participants could interact with the tool, mirroring real-world usage as closely as possible and allowing them to provide feedback on their experience.

To describe the environment, Figure 3.1 illustrates the CodeQL tool, project directories, and the warning generated by the static analysis tools.

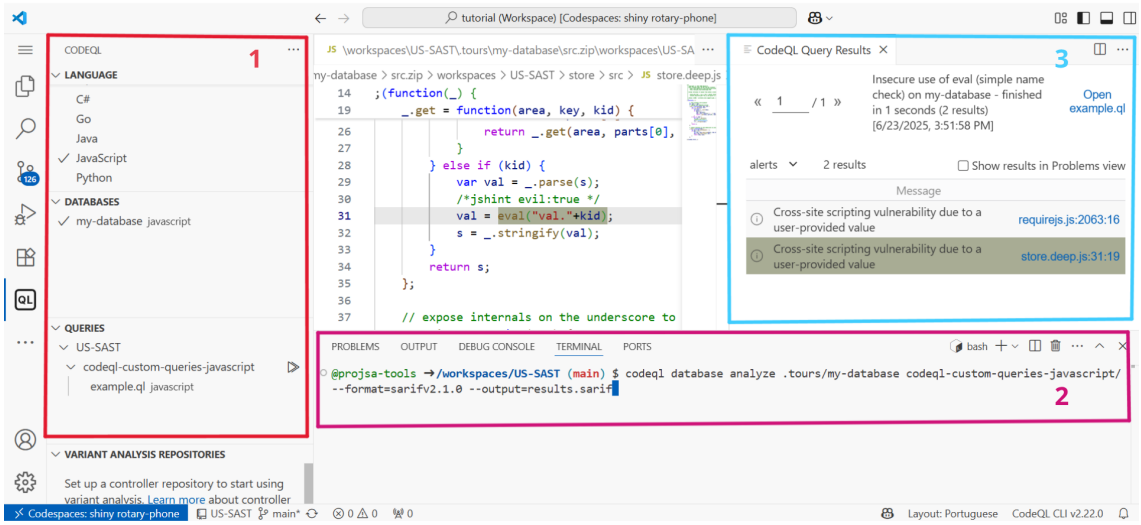

Figure 3.1: Environment Setup

The Figure 3.1 shows 3 different sections:

- The section 1 (red section) shows the CodeQL extension. It is possible to see the language, database and queries configuration sections.
- The section 2 (pink section) shows the command that allows the code to be scanned by the tools.
- The section 3 (blue section) shows the alerts generated by CodeQL for the query executed.

To run CodeQL, it is necessary to create a codebase to be used by the static analyzer. The following command is executed for this purpose:

```
codeql database create .tours/my-database --language=javascript --
    source-root=./store
```

Once the database had been created, the tool must be configured, starting with the choice of language (JavaScript, in this case), followed by the selection of the database (`my-database`), and finally running the query to trigger the alert.

3.2.1.1 Case Study Vulnerability

To ensure real-world relevance, the vulnerability used in this study was selected from the Open Source Vulnerability (OSV) database. This is a distributed vulnerability database that aims to provide a standardized and open-source database with accurate and up-to-date information. It supports various ecosystems, including npm (Node.js), PyPI (Python), and Maven (Java). This infrastructure serves as an aggregator of reliable vulnerability database sources [OSV, 2025].

The chosen vulnerability affects the *nbubna/store* package and was published on January 23rd, 2025. It is identified as CVE-2024-57556, classified as a medium-severity (6.1) cross-site scripting (XSS) issue [GitHub, 2025b]. The vulnerability was fixed on December 24, 2024.

This vulnerability stems from the use of the `eval()` function, which executes a string as JavaScript code. When the argument passed to `eval()` originates from user input or other untrusted sources, it introduces a significant security risk—enabling potential code injection and consequently remote code execution. The relevant and vulnerable code fragment is shown below:

```
...
else if (kid) {
    var val = _.parse(s);
    /*jshint evil:true */
    val = eval("val."+kid);
    s = _.stringify(val);
}
```

As stated before, the problem in this fragment of code is the use of `eval()`. The use of this function is considered bad practice and a risk. In addition, it is a slow function and makes code maintenance difficult, as it makes reading and debugging more complicated. This function allows arbitrary code execution, since it executes strings as code, and the `kid` parameter, passed

to the function, is not sanitized and can come from an untrusted source. The line `eval("val" + kid)` is being used to access the properties of an object, val, but this is an unsafe and inefficient way of doing it. In this sense, the fix would be to replace `eval()` with a safe and direct way of accessing properties, for example, using the bracket notation `val[kid]`.

This specific vulnerability was selected for the following reasons:

- **Language accessibility:** The package is written in JavaScript—a language widely used in web development and familiar to most developers.
- **Repository accessibility:** The code repository and relevant commit history (including the vulnerable and fixed versions) were publicly available, allowing realistic integration into the experiment environment. Not all published vulnerabilities have accessible repositories.
- **Educational suitability:** Cross-site scripting is a well-known vulnerability type that is relatively straightforward to understand and fix, making it appropriate even for developers with limited security experience. This vulnerability is part of the OWASP top 10 and occupies the third place in the list, which makes it one of the most frequent and critical occurrences currently [OWASP, 2025a].

3.2.1.2 Static Analyzer

The static analysis tool integrated into the experimental environment was CodeQL, an open-source semantic code analysis engine developed by GitHub. CodeQL enables developers and researchers to write custom queries that scan source code for patterns associated with bugs and security vulnerabilities. It is widely used in industry and academia and provides access to an official repository of pre-built queries covering various vulnerability classes [GitHub, 2025a]. This static analyzer uses a data flow analysis technique to compute the possible values that variables can have during the program, and determines how they can be propagated and where they are used [CodeQL, 2025]. In this case, this factor is important in identifying the chosen vulnerability to recognize how the `kid` flows to the `eval()` function.

CodeQL was selected for the following reasons:

- It is a well-established tool, commonly used in both industry and security research [Khare et al., 2025, Ayala and Garcia, 2023, Chapman et al., 2024].
- It supports extensibility through custom queries, enabling precise control over the types of vulnerabilities being detected.
- It can be used on different traditional programming languages.

During the initial setup, however, it was found that none of the existing queries for JavaScript cross-site scripting (XSS), also known as CWE-79, successfully defined the vulnerability present in the *nbubna/store* package. The official repository for JS has seven rules for XSS vulnerability[1]. These rules cover some of the main XSS attack vectors:

[1] https://github.com/github/codeql/tree/main/javascript/ql/src/Security/CWE-079

- *XSS DOM-based*: This is a client-side attack, i.e, the malicious script is not executed on the server. The attacker directly manipulates the DOM in the browser. In this case, the queries involved directly inserting the code into the DOM and reinterpreting text as HTML in the DOM.
- *XSS Reflected (non-persistent)*: The attacker tricks the victim into clicking on a malicious link, where the server reflects the value in the HTTP response with the user input requested.
- *XSS Stored (persistent)*: The malicious script is stored on the server and then displayed to others without sanitization.

To address this lack, a custom query was developed to detect the use of the `eval()` function—a well-known XSS sink when applied to user-controlled input:

```
/**
 * @name Insecure use of eval (simple name check)
 * @description Flags any function call whose callee is named "eval"
 * @kind problem
 * @problem.severity warning
 */
import javascript

from CallExpr call
where call.getCalleeName() = "eval"
select call, "Cross-site scripting vulnerability due to a user-provided
    value"
```

This query searches for all the function calls to the function `eval()` and flags them as an alert. The warning message is "Cross-site scripting vulnerability due to a user-provided value".

3.2.2 Pre-Screening Survey

This section describes the first phase of the study: the design and implementation of the pre-screening questionnaire. The purpose of this phase was to collect and analyze data about the background and prior knowledge to define participants' profiles.

3.2.2.1 Design

The Pre-Screening questionnaire (Pre-Screening Form) was developed to introduce the study and collect relevant background information from prospective participants. The form contained 17 questions, organized into the following sections:

- **Overview of the Study:** A brief explanation of the study's purpose, objectives, methodology and subsequent phases.
- **Consent:** A mandatory section outlining the study's terms and conditions. Participants were required to provide informed consent to proceed.

- **Background and Experience:** Questions assessing programming experience and exposure to software security knowledge. These questions serve to gather insights into the participants' level of expertise in programming and security, which is important for defining participants' skills. and profiles.
  - Example: *How many years of experience do you have in software development? Have you ever worked on security-related aspects of software development?*
- **Confidence with Security Issues:** Questions designed to gauge participants' self-assessed confidence in identifying and resolving security-related issues. These questions are intended to gauge the participants' level of comfort in dealing with warnings and fixing vulnerabilities in order to analyze their autonomy and confidence in interpreting and resolving alerts.
  - Example: *How confident are you in understanding what caused a security warning and how it could be exploited? How confident are you in handling security warnings in code?*
- **Use of SAST:** Questions exploring prior exposure to SAST tools. These questions collect the factors that lead participants to make decisions about warnings and the interaction they have had with SAST tools in order to ascertain developers' opinions about static analyzers and their limitations.
  - Example: *What factors make you more likely to ignore a security warning? When you see a security warning in code, what do you usually do first? Have you used static analysis before?*
- **SAST Experience and Trust:** An option section for participants with prior SAST experience, focusing on perceptions and trust in these tools. These questions provide insights into the participants' experience of using these tools and the level of trust they place in them.
  - Example: *Have you ever encountered a false positive from a static analysis tool for a security issue? How much do you trust security alerts generated by static analysis tools? Have you ever found a static analysis warning helpful in understanding or fixing a security issue?*

The questionnaire included a variety of question types, such as multiple choice, checkboxes, Likert scales, and one open-ended question.

3.2.2.2 Protocol

The survey was implemented using Google Forms and distributed via email to participants from both academic and industry backgrounds. Participation was voluntary, and no personally identifiable data was collected, aside from an email address used solely for scheduling the next phase.

The survey was completed asynchronously and took approximately 10-15 minutes to complete. Responses were collected over a one-month period. Participants who consented and

matched the study criteria were invited to continue to the next phase. The criteria for participants to be included in the study were that they had some programming experience and a background in software development. In total, 12 valid responses were received, all of whom proceeded to successfully schedule and participate in the interviews.

3.2.3 Hands-On Experiment

The second phase of the study consisted of individual interviews with the 12 participants who completed the pre-screening phase. Each participant was asked to perform a security-focused task, allowing for direct observation of their reasoning and problem-solving process. All interviews were conducted remotely via Google Meet, with both screen sharing and audio enabled to support session recording. Prior to each session, participants were reminded of the consent terms—specifically, what data would be recorded—and were asked to confirm their agreement once again.

The task took place within the experimental environment described in Section 3.2.1, where participants interacted with a real-world vulnerable codebase. After generating an alert using CodeQL, participants were asked to interpret the warning and attempt to resolve the underlying issue using any resources they deemed appropriate, including web search engines, documentation, or AI assistants. The interviews followed a think-aloud protocol: participants were instructed to verbalise their thoughts as they explored, interpreted, and addressed the reported vulnerability. Figure 3.2 shows the GitHub Codespace environment used during the experiment, highlighting both the alert and the corresponding code fragment that triggered the alert.

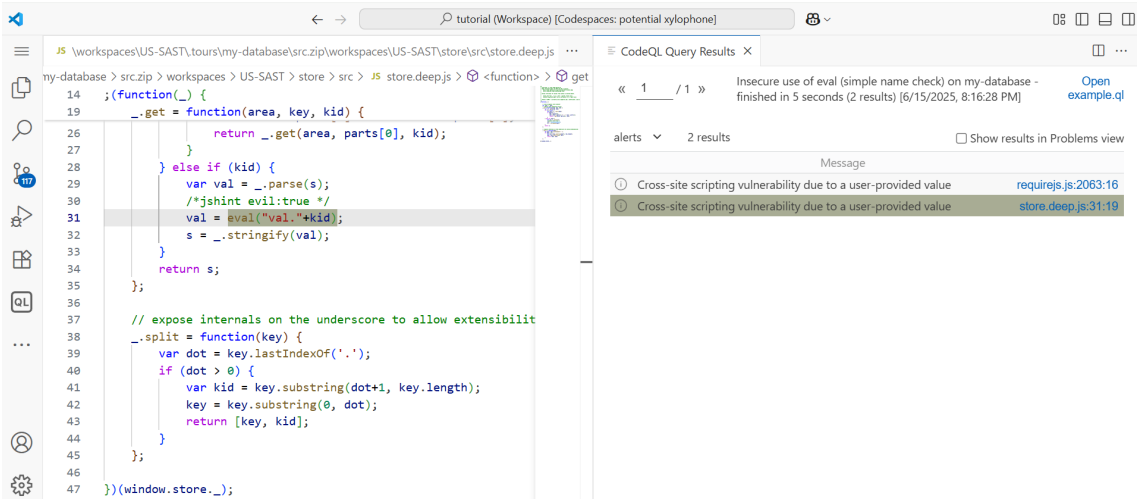

Figure 3.2: GitHub Codespace environment with the vulnerable code and the alert

Throughout the sessions, interviewers took observational notes on the following observations:

- How participants reacted upon encountering the alerts (e.g., whether they engaged immediately or hesitated);
- What actions did the participants take to understand and fix the vulnerability;

- Which external sources they consulted (e.g., documentation, search engines, AI tools);
- Signs of frustration, confusion, or skepticism while performing the task;
- Time taken to initiate action and complete the task.

Recordings were later transcribed using TurboScribe, an AI-powered transcription tool. Each recording and its corresponding transcript were then manually reviewed. This process focused on accurately capturing participants' verbalized reasoning, problem-solving approaches and use of external support.

To validate the experimental setup, a pilot session was conducted with two participants not included in the main study. One of the sections revealed a critical issue: the main code file was in read-only mode, which prevented the participant from editing and validating their fixes. Despite this obstacle, that pilot session was completed, and the participant's solution was successfully assessed. Further investigation revealed that the issue was caused by the use of a submodule reference for the original repository within GitHub CodeSpaces, which restricted write access. The problem was resolved by replacing the submodule file with a new editable copy of the original content. The other pilot session proceeded as expected, from the setup of the environment to the completion of the task. During the preparation of the interview script, several follow-up questions were created to address potential moments of blockage and silence, offering some guidance when needed - for example, to clarify the reasoning involved in understanding the alert through questions such as "What are you thinking right now?" and "What information would help you at this point?". This session allowed for the testing of those set questions and helped assess their effectiveness in encouraging participants to engage naturally with the think-aloud protocol.

3.2.4 Post-Interview Survey

This section describes the final phase of the study: the design and administration of the post-interview questionnaire. The purpose of this phase was to consolidate qualitative insights gathered during the task with structured post-task feedback regarding the experience and intellectual pressure.

3.2.4.1 Design

The Post-Interview Survey (Post-Interview Form) was developed to gather participants' reflections on the task, including the perceived challenges, reasoning strategies and overall experience. The questionnaire consisted of 17 questions, organized into the following sections:

- **Overview of the Study:** A summary of the study's objectives and the interview protocol.
- **Consent:** A mandatory section outlining the terms and conditions of participation. Informed consent was required to proceed.
- **Protocol:** A short description of the task, including its goals and expectations.

- **Post Study Survey:** A set of questions aimed at assessing participants' interpretations of the alert, reasoning processes, challenges encountered, and reliance on external resources (e.g., documentation or AI tools). This section was further organized into thematic areas:
  - **Emotional response and comprehension:** Questions exploring participants' feelings while interpreting the alert and their understanding of the message. These questions aimed to assess message clarity, perceived usefulness, and the emotional impact of the alert—such as frustration, confidence, or confusion—which provide indirect indicators of usability, trust, and cognitive effort.
    - Example: *How clear was the alert to you? How did you feel reading the report? What was the security issue CodeQL was warning you about?*
  - **Actions and resolution strategies:** Questions addressing the actions taken to respond to the alert, the rationale behind chosen fixes, and the external resources used. These aimed to capture behavioral patterns in vulnerability resolution.
    - Example: *If you found a similar security alert in your daily work, what would you do? Did you attempt to fix the issue?*
  - **Perceived barriers to comprehension:** Questions identifying factors that made the alert more difficult to understand, aiming to uncover common pain points in alert communication.
    - Example: *Which of the following made understanding the alert most difficult? Which changes would you perform to simplify the task?*
- **NASA TLX (NASA Task Load Index):** An adapted version of NASA's workload assessment tool [2] was used to measure participants' perceived cognitive load and stress during the task. Due to limitations in Google Forms, the original 21-point scale was adapted to a 10-point scale.

The questionnaire featured a variety of question formats, including open-ended responses, multiple-choice questions, linear scales, and checkboxes.

3.2.4.2 Protocol

The survey was implemented using Google Forms and administered at the end of each interview session. All participants completed the form during their scheduled Google Meet call. No personal data was collected. Instead, each participant was assigned a unique Participant ID, which was used to maintain anonymity. The form took between 20 and 30 minutes to complete. All participants who reached this phase completed the survey in full, ensuring consistency across responses and contributing to the internal validity of the study.

[2]The NASA TLX is a widely used workload assessment tool developed by NASA's Ames Research Center. It evaluates mental demand, effort, frustration, performance, and task pace [Center, 2020].

3.2.5 Data Protection

Given the nature of this study, appropriate measures were taken to protect participants' privacy and ensure ethical data handling throughout all phases. In this sense, the study complied with institutional data protection policies and ethical guidelines, and all procedures were submitted to the Ethics Committee of FEUP.

No personal data were collected beyond what was strictly necessary for study coordination. The only identifying information requested was an email address in the Pre-Screening form, used solely for scheduling interviews. Both the Pre-Screening and Post-Interview questionnaires included a consent section outlining the study's terms and conditions, including the use of audio and screen recordings. Participants who declined consent or recording would have been excluded from further participation, and any data collected up to that point would have been securely deleted. In addition to this, participants retained the right to withdraw at any stage without penalty, and to request the removal of their data at any point prior to final analysis. However, all participants provided informed consent and completed the study in full.

To ensure anonymity, each eligible participant was assigned a unique alphanumeric identifier following the Pre-Screening phase. This identifier was used consistently to label all files—including screen recordings, audio transcripts, and questionnaire responses—ensuring that no personally identifiable information was stored or linked to the data.

All study materials were securely stored in a Google Drive folder, with access restricted to the research team. File naming and storage practices adhered to the established identifier protocol to maintain confidentiality and ensure traceability throughout the analysis process.

Upon completion of the project, all raw data (e.g., screen recordings, audio files, and unprocessed survey forms) will be securely deleted. Only anonymized excerpts—such as selected quotes from transcripts, aggregated survey results, and summary metrics—will be retained for dissemination and included in publicly available replication packages. This data will be retained for 3 years for audit and reproducibility purposes.

3.3 Participant Background and Profiles

Participant profiles were derived from responses to the Pre-Screening questionnaire. These data allowed us to analyze whether programming experience, security exposure, or familiarity with static analysis tools influenced participants' ability to resolve the assigned task and the kind of external support they relied on.

In total, 12 participants (IDs USP01 to USP12) completed the Pre-Screening phase and proceeded to the interview stage. This section summarizes their backgrounds and experiences.

3.3.1 Background and Security Experience

This section presents participants' backgrounds, focusing on their software development experience, exposure to security-related tasks, and familiarity with assessing software vulnerabilities.

Development Experience. The 12 participants ranged from junior to senior developers, with experience spanning from *less than 2 years* to *over 6 years* in software development. Table 3.1 presents the complete distribution of participants according to their years of experience in development. The sample includes university students, academic researchers, and industry professionals.

Software Development Experience	Number of participants
Less than 2 years	1
Between 2 and 4 years	1
Between 4 and 6 years	4
More than 6 years	6

Table 3.1: Number of participants per years of experience

Security Exposure and Familiarity. When asked about *past exposure to security-related aspects of software development*: **4 participants (33%)** had *direct experience* with secure coding or security-focused tasks, **6 participants (50%)** had *indirect exposure* (e.g., encountering security issues during development), and **2 participants (17%)** had *no relevant exposure*. In terms of the *frequency of encountering security vulnerabilities*, responses ranged from "never" to "very often": **2 participants** reported *never* encountering such issues, 4 participants reported *rarely* encountering them, **4 participants** reported *sometimes* encountering them, **1 participant** reported *often* encountering them, and **1 participant** reported *very often* encountering them. Only **2 participants (17%)** reported encountering vulnerabilities frequently (often or very often), suggesting that most are not regularly engaged with security-related tasks. This aligns with their roles as generalist developers who likely face security issues incidentally—e.g., during feature development or code reviews—rather than through dedicated security responsibilities.

Participants' self-reported familiarity with common security vulnerabilities ranged from *not at all* familiar (**1 participant**) to *extremely familiar* (**1 participant**), with the majority (**7 participants**) describing themselves as *moderately familiar*. Familiarity generally correlated with exposure: 1) among the **8 participants** who reported *some degree of familiarity*, **6** had encountered security issues regularly; 2) In contrast, the **3 participants** with little to *no exposure* also reported *minimal familiarity*. This suggests that participants were neither complete novices nor security specialists. Most had moderate familiarity with vulnerabilities, primarily shaped by their past exposure to security issues.

3.3.2 Confidence in Understanding vs. Acting on Alerts

When asked about their *confidence in understanding the cause of security problems and how they might be exploited*, most participants reported a *moderate to high level of confidence*, typically rating themselves between 3 and 4 on a 5-point scale. Only one participant indicated no confidence

in this area. However, **confidence dropped noticeably** when participants were *asked about their confidence in actually solving such issues*. Responses were generally lower, reflecting greater uncertainty and skepticism regarding their ability to implement effective fixes. These findings indicate that while most participants felt capable of understanding the root cause and potential impact of the security issue, they were significantly less confident in resolving it.

3.3.3 SAST Usage and Trust

This questionnaire also served to validate initial assumptions about participants' perception of static analyzers for security, particularly their attitudes toward the alerts these tools produced. These pre-task opinions were later compared with the participants' reflections during the interviews. Figure 3.3 shows the *main reasons developers ignore static analysis alerts*: perceived false positives (23.3%), time pressure (20%), and low alert impact (16.7%). Other contributing factors were unclear or overly technical messages (13.3%), lack of explanation from the tool (13.3%), uncertainty about how to fix the issue (10%), and assuming someone else is responsible (3.3%). Figure 3.4 shows **what developers do when first encountering a static analysis alert**: try to understand what it means (31.6%), followed by searching online or consulting documentation (21.1%), and attempting a fix and testing it (18.4%). Others either consider the context before acting (15.8%) or check if the alert is a false positive (13.2%).

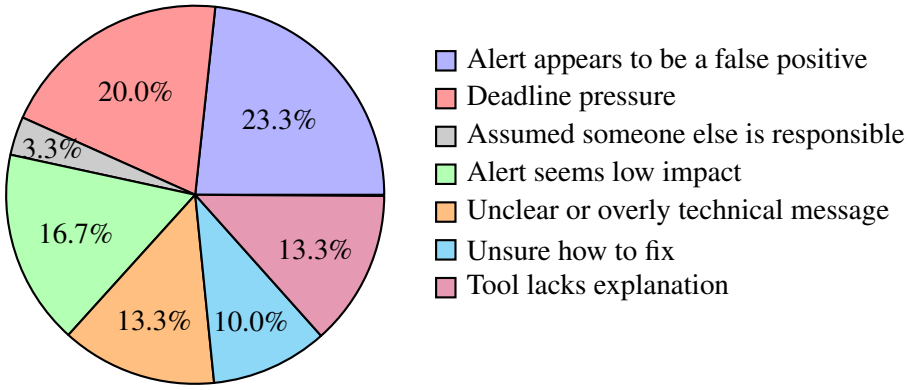

Figure 3.3: Reasons developers might ignore static analysis alerts

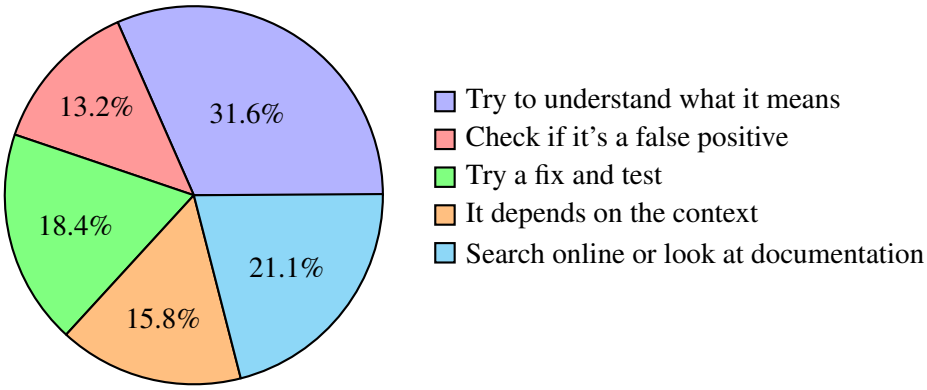

Figure 3.4: Typical first reactions to static analysis alerts

Only **33% of the participants** reported prior experience with static analysis for security, while the rest had either not used them or were unsure. The previously used tools were the following (some participants used more than one of these tools): CodeQL (60%), Bandit (20%), SonarQube (40%), ESLint with security rules (40%), and IntelliJ's IDEA built-in static analyzer (20%).

Among those with experience with SAST, all expressed trust in the accuracy and usefulness of the alerts, noting that the tools had helped them understand and fix issues in their code. Participants who were unsure whether they used these tools demonstrated some hesitation and mistrust. One of them noted that the tools seem useful, while another remained uncertain about the usage. Those who had no experience with the tools did not express any opinion regarding the tools' trustworthiness or helpfulness.

When *faced with uncertainty about specific warnings*, **participants commonly turned to external resources for support**, such as official documentation, forums such as `StackOverflow`, AI-based tools such as `ChatGPT` or `GitHub Copilot` and GitHub issues or pull requests. Participants identified several reasons why they found the static analysis warning experience helpful:

- **Seamless integration with existing developer workflows**: *The tools can be easily and directly integrated into the other tools that developers use, for example, SonarQube and CodeQL can show errors in the GitHub UI, while ESLint shows errors alongside other linting errors.* – by Participant USP05.
- **Effective detection of security issues**: *These tools detect security issues in big codebases and big pull request changes, which developers are often unaware of.* – by Participant USP05.
- **Educational value**: *Vulnerabilities that are pointed out help to correct problems in the code and serve to teach and identify them for future projects.* – by Participant USP03.

3.3.4 Participant Profiles Summary

Based on the data collected in the pre-screening questionnaire, participant profiles were categorized by their programming experience, exposure to security-related issues, self-reported confidence in dealing with security problems, and prior use of static analysis tools. Table 3.2 summarizes these dimensions, along with perceptions of SA tool trustworthiness and usefulness, and the external resources participants relied on when interpreting or resolving alerts.

This characterization reveals a participant group that reflects a realistic cross-section of generalist developers. While they differ in terms of professional background and experience, most have had at least some incidental exposure to security-related issues, though few are security specialists. Their varying levels of confidence—particularly the distinction between understanding a vulnerability and knowing how to fix it—suggest that they represent the target audience for tools that aim to bridge the gap between detection and actionable remediation. The range of external resources they rely on, including AI tools and community forums, also highlights a clear demand for clearer, more contextualized support when engaging with security alerts.

Experience	Security Expose			Confidence		Used SA tools?	Find them		External Tools
	Have?	Frequency?	Familiarity?	Understanding	Handling		Trusty?	Helpful?	
< 2 years	No	Never	Not at all	Very	Very	Not sure	-	-	Documentation AI tools
2 to 4 years	No	Sometimes	Moderately	Medium	Medium	Not sure	-	-	-
4 to 6 years	Not direct	Rarely	Slightly to Very	Medium	Medium	Yes (one)	Yes	Yes	StackOverflow GitHub PR AI tools
4 to 6 years	Yes	Sometimes to Often	Moderately	Confident	Confident	Yes	Yes	Yes	Documentation CWE databases
> 6 years	No	Rarely or never	Slightly to Moderately	Low	Medium	Yes (one)	Yes	Yes	Documentation StackOverflow GitHub PR AI tools
> 6 years	Yes	Sometimes to Very Often	Moderately to Extremely	Confident	Confident	Yes	Yes	Yes	Documentation CWE databases StackOverflow

Table 3.2: Participants' profiles

Chapter 4

Results and Discussion

This section begins by presenting the results of the study, and the subsequent conclusions on the data collected are analyzed and discussed.

4.1 Results

Following the responses to the Pre-Screening questionnaire, 12 interviews were scheduled and conducted: two were conducted in English and the remaining ten in Portuguese.

A combination of qualitative and quantitative methods was used to analyze the data collected during the interview phase, in order to understand the programmers' thought processes, identify their challenges, and assess the effectiveness of LLMs in supporting their tasks.

The questionnaires collected data on common pain points and contextual factors that influence programmers' workflows. While the observational data made it possible to categorize programmers' tasks, highlight common bottlenecks and map areas where LLMs can provide significant assistance.

4.1.1 Behavioral Patterns (RQ1)

This section addresses the first research question, which explores how developers behave and make decisions when solving alerts from static analysis tools. It presents findings based on observed behavioral patterns as participants engaged with the task and the chain of thought followed by them, offering insights into how they interpreted and responded to alerts.

4.1.1.1 Common Patterns

At the beginning of the task, all participants immediately read the warning as soon as it appeared on their screens and clicked the reference to the line that triggered it (line 31 in the *store.deep.js* file).

After reading the report, most of the participants recognised the vulnerability, and three were able to explain what XSS entails.

While 25% of participants immediately used AI tools for support in explaining and solving the alert, the remaining 75% initially attempted to explore and analyze the code manually. Among these participants, 33% first tried to understand the alert by examining the line that caused the problem. They investigated the use of `eval()` function and tried to trace the origin of the parameter (`kid`) passed to it, before concentrating on the broader context of the code. The other 67% took the opposite approach, first trying to understand the general structure and purpose of the code before attempting to solve the problem. In this initial part, some participants appeared to be somewhat lost and unsure of how to start. They stated multiple times that the code was confusing and difficult to interpret. As a result, some of these participants were unable to have a clear view of the program and eventually shifted their focus to solving the problem without fully grasping the underlying code.

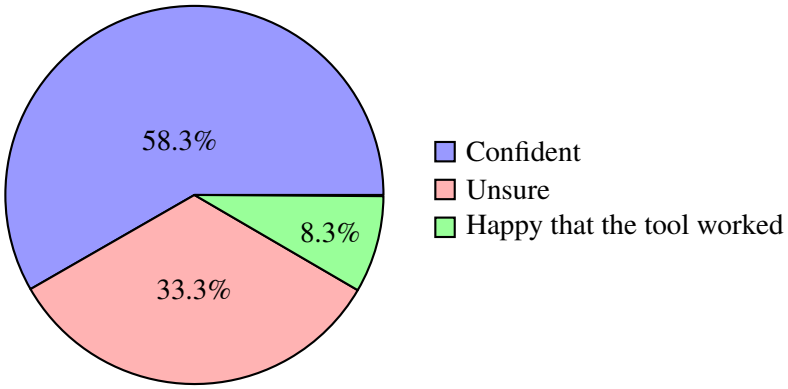

Figure 4.1: Feelings reported while reading the report

Regarding the warning, most of the participants ended up interpreting that the cause of the XSS identified was that the parameter (`kid`) passed to the function might come from an untrusted source, and there was no prior validation of the value. A few participants explored the output more carefully and noticed the comment *"Insecure use of eval"* near the beginning, which led them to suspect the `eval()` function as the root cause. Whereas some were already familiar with the function, only one recognized its unsafe nature. The rest only realized its risks after further investigation.

After understanding the alert and possible exploitation of it, practically all the participants acknowledge the vulnerability as a serious security risk with potentially significant consequences. One participant affirmed, *"The risk of ignoring important vulnerabilities may represent an high cost for the organization responsible for the codebase"*. While another participant expressed the impact of such exploitation, *"Cross-site scripting is a severe vulnerability. For instance, if I'm developing a library, a cross-site scripting vulnerability can make all consumers of the library vulnerable. If I'm developing an application, the application itself can become vulnerable. Vulnerable scripts can then be exploited to extract cookies from browsers, keylog users and/or impersonate users, among other attacks"*.

As a result, the majority said they would prioritize or take immediate action to solve the alert in a real-world scenario. However, their justification and opinions varied slightly. One participant

said it would depend on the situation, *"It depends of the application domain. But, if security is a critical requirement, and the system does not crash thanks to the issue, so I chose prioritize"*. While other considered the vulnerability less urgent, *"This issue is valid but not critical. It can wait some time for a fix. It also occurs in the functionality, which is considered 'alpha', so users expect less quality from it. I would create a ticket/issue and let my team know about this vulnerability. I would also advise my team (and the users of our 'store' library, if any) to not use this feature which depends on 'eval' because it's insecure"*.

Regarding the fix, 33% of the participants believed that the solution included sanitization or validation of the input `kid`. Only after verifying that the warning persisted did they suspect that the root of the problem might be the use of `eval()`. The rest turned to external resources, namely AI tools, for assistance in fixing the issue. Of these, only one made a critical analysis of the fix suggested by the chosen tool. The others relied on the proposed solution without deeper evaluation. It is worth noting that three participants perceived the role of `eval()` in the context of the program. They only used external resources because they were unfamiliar with how to access object properties in JavaScript.

Three participants did not use AI to accomplish any of the parts of the task. They indicated that they already knew the function, but searched external resources to confirm its behavior. They successfully understood the vulnerability and its cause only through documentation or Internet searches. As for implementing the fix, two of them understood what the purpose of the code and researched how to access elements in JavaScript. The third one completed the task without any external support. The reason for not using AI were the following:

- Preference for solving the problems alone, *"When it comes to security issues I prefer to not rely on AI tools as I would like to understand the problem myself and come up with my own solution"*.
- Preference for obtaining information from official documentation, *"First, I didn't know the eval function and I prefer to go directly to its documentation"* (translated from PT to EN).
- Considered AI was not necessary, *"After doing that [search for eval description], the vulnerability seemed obvious to me. I didn't think AI was necessary. If it were a more obscure issue, then I would have used it"* (translated from PT to EN).
- Lack of quality in results in tools not integrated in the environment, *"I think the quality of non-integrated AI tools without context access — like ChatGPT — has gotten worse when it comes to code. If that environment had Copilot, or if we were using Cursor, and if I thought it was necessary, then I would have used it"* (translated from PT to EN); *"I didn't use AI tools like ChatGPT because my impression is that these tools require a lot of context to properly identify the issue and come up with a solution. For example, I'd need to copy-paste the error, the relevant code, and explain the situation in which the error occurred (based on a CodeQL analysis). And from my initial impression of the problem, it seemed like something simple to fix"* (translated from PT to EN).

At the end of the task, 92% of the participants explained the security issue in their own words. This written explanation served as an indicator of whether they had truly managed to understand and clarify the warning. In the responses, it emerged that not all participants fully grasped why the alert had been triggered. For example, one broadly described the vulnerability as *"A possible point for cross-site scripting attack"*; and other one concentrated on a general assumption *"CodeQL is warning me about XSS. This usually is tied with not sanitizing the input that comes from the user"*.

4.1.1.2 Chain of Thought (CoT)

After reviewing the interviews, it was possible to extract common behavioral patterns. So, a general chain of thought characterized by the following steps was defined:

- **Step 1 - Understanding the Warning:**
  - Carefully read the alert;
  - Redirect the tool to the line indicated in the alert;
  - Identify the vulnerability.
- **Step 2 - Analyzing the general context of the code:**
  - Try to understand the execution flow and the purpose of the code.
- **Step 3 - Determine the action to take:**
  - Clearly understand what the line that triggered the issue was doing;
  - Investigate whether it is a problem that should be resolved immediately or not.
- **Step 4 - Fix the problem:**
  - Think or search for the best way to fix the problem safely;
  - Guarantee that the code maintains original functionality (through code review).
- **Step 5 - Validate the fix:**
  - Check if the alert disappeared after executing the static analysis tool;
  - Run automatic tests on the code.

In the interviews, 67% of the participants followed the entire chain of thought described, whereas the remaining 33% only followed it partially. Table 4.1 describes the chain of thoughts extracted from the participants' actions during the task. The **67% of participants** who followed all the steps adhered to **CoT1**, while the remaining participants were divided among the other chains identified. The most skipped phases were *Step 2* and *Step 3*, by participants who opted to use AI tools immediately (**CoT4**). Additionally, one participant discarded the validation phase, arguing that confirmation was unnecessary since the problem clearly stemmed from the use of

`eval()`, and replacing it would eliminate the issue (**CoT3**). Lastly, the participant who followed **CoT2** skipped *Step 3*, hesitating after analyzing the code and processing directly to request a fix from an external tool, rather than determining an appropriate course of action on their own.

Chain of Thought	Step 1 (Warning understanding)	Step 2 (Context analysis)	Step 3 (Determine action)	Step 4 (Fix issue)	Step 5 (Validate solution)	CoT adherence
CoT 1	Y	Y	Y	Y	Y	67%
CoT 2	Y	Y	N	Y	Y	8%
CoT 3	Y	Y	Y	Y	N	8%
CoT 4	Y	N	N	Y	Y	16%

Table 4.1: Chains of Thought extracted

4.1.1.3 Subjective Workload Assessment

To further support the analysis of the participants' performance, their self-assessment during the interview phase was incorporated, using the NASA TLX standard described in the previous chapter. The Table 4.2 presents the rating of each participant across the five dimensions considered (Mental Demand, Temporal Demand, Performance, Effort and Frustration Level), along with the time they took to complete the task.

Participant	Mental Demand	Task pace	Success in task	Effort required	Negative feelings	Duration
USP01	6	4	3	7	2	19:33
USP02	6	8	2	4	5	24:30
USP03	5	3	3	4	4	18:26
USP04	7	5	3	7	8	15:40
USP05	1	1	2	3	1	17:34
USP06	3	2	2	2	1	13:50
USP07	7	5	2	5	2	05:04
USP08	2	2	3	2	5	16:15
USP09	5	1	1	5	1	07:57
USP10	7	5	-	7	6	24:46
USP11	5	8	4	6	1	14:04
USP12	7	6	3	7	5	18:28
Average	5.1	4.2	2.3	4.9	3.4	16:34

Table 4.2: NASA TLX responses

Based on the data collected in the table above, the conclusions can be broken down into the various fields of evaluation.

Starting with mental demand, the participants found the task moderately demanding in terms of the reasoning required. This was an expected result, since code comprehension tasks require analytical thinking and concentration.

With regard to the pace of the task, the answers suggest that participants felt a little pressured, despite the fact that no time limit had been imposed for completion, nor was the task being timed.

Related to the perception of their performance in completing the task, the participants generally believed that they had successfully completed the challenge. An interesting aspect to highlight is that although the responses were positive, at the end of the interviews, when they were asked if they trusted their solution, some of those who rated themselves with the highest score showed insecurity, *"my lack of expertise in the area makes me not so confident, even though I resolved it and CodeQL passed, before committing this, I'd really need to understand what it actually does"* (translated from PT to EN with ChatGPT); doubts whether they had managed to solve the problem without damaging the correct execution of the code, *"in terms of protection against XSS, I think so, well, kind of, I'm not sanitizing. But if it were an XSS attack, I think it would throw an error at that line. So yeah, I think it's fine. But when it comes to whether the program actually runs correctly, I do have my doubts"* (translated from PT to EN with ChatGPT); or distrust in the tool used, *"the fact that it disappeared isn't 100% reliable, because I'm not sure there isn't another flaw, since we're using AI it comes up with some answer just to satisfy. It's possible we still have another issue or error. I'd need to test it more"* (translated from PT to EN with ChatGPT).

Moving on to the effort required, the evaluation indicates that the participants felt they had to put a reasonable amount of effort into completing the task. This is probably due to the fact that many spent a lot of time understanding the code, but in terms of solving the issue, most used external tools. The majority of the highest responses were from participants who tried to carry out all the steps themselves.

The last topic assessed was the feeling of anxiety or frustration experienced while solving the task. The values were relatively low (3.4/10), so apparently the task did not cause much stress in the participants. However, at the beginning, some of them showed signs of insecurity and confusion when trying to understand the code. This is shown by the fact that it took them between 20 to 34 seconds to take any action to start the task and they asked for guidance at the beginning through questions such as *Now I have to understand the code?*, *So now I have to try to understand it for the vulnerability to disappear, is that it?* and *Is there no other documentation in this repository?*. Finally, the participants who tried to solve the warning on their own had the highest stress and frustration scores.

4.1.2 Cognitive Challenges (RQ2)

This section addresses the second research question, in which the aim was to identify the main difficulties and common pain points in understanding and solving the output of static analyzers.

Insights were gathered by observing participants' use of external resources, as well as registering the questions they asked during the task.

4.1.2.1 Main Challenges

Starting with analyzing the greatest difficulties observed during the interviews, it was possible to highlight the following:

- **Understanding what the `eval()` function does:** when the participants suspected that the vulnerability might be caused by the `eval()` function, the majority were unaware of how it worked or the risks of using it. So, to overcome this situation, they searched for it in search engines (Google Chrome, DuckDuckGo), documentation, and AI-based tools.
- **Understanding the context of the code and its purpose:** As mentioned in the previous section, almost all the participants tried to understand what the code did in general, but it was a challenging task as it was the first time they had seen the code and according to them it was written in a confusing and unnecessarily complex way. Many said it was important to understand the code before trying to fix the vulnerability, but others, after analyzing it for some time, simply went ahead and fixed the alert. Interestingly, despite the difficulties they were having, no one asked for outside help in understanding the code. The participants who managed to reach a conclusion about the program did so on the basis of their manual analysis and review of the file and repository.
- **Realizing what the function was being used for:** Some participants struggled to figure out the role of the `eval()` function in the context it was in. However, once again, no one actively sought an explanation, just assuming that what they were thinking, even though they were not sure, or giving up and asking AI to help them solve the problem reported in the alert.
- **Understanding how to solve the problem:** Almost all participants did not know how to replace `eval()` with a safer alternative. This may have been compounded by the fact that they did not understand what the function was being used for. Many asked for help from AI tools (ChatGPT, DeepSeek) to provide them with a solution to the warning. There was also the case of participants who realized the purpose of the function in the context, but did not know how to access object properties in JavaScript and so searched on search engines to find the right way to do it.

In addition to these difficulties encountered by the participants when carrying out the task, another factor that was investigated was the faults pointed out in the report, which, in the participants' view, aggravated the obstacles to completing the task. It was found that although they considered the warning to be clear (average rating of 8.53 out of 10), practically all needed external help to supplement the information provided in the alert or to help them actually fix the issue. In this sense, although the general opinion of the alert was positive, the participants mentioned aspects of the alert that hindered their understanding and immediate resolution, namely:

- The alert was vague and lacked context, *"It is very broad, it points to a potential issue with no clear context or ways to address it"*.
- No description of the `eval()` function, *"however it didn't describe exactly what the eval() function did so it required an internet search to actually understand the problem with using the eval() function"*.
- Lack of resolution guidance and examples of fixes, *"I had no idea of how to solve"*.

To supplement this information, Figure 4.2 shows the participants' responses when asked which factor made it most difficult for them to understand the alert.

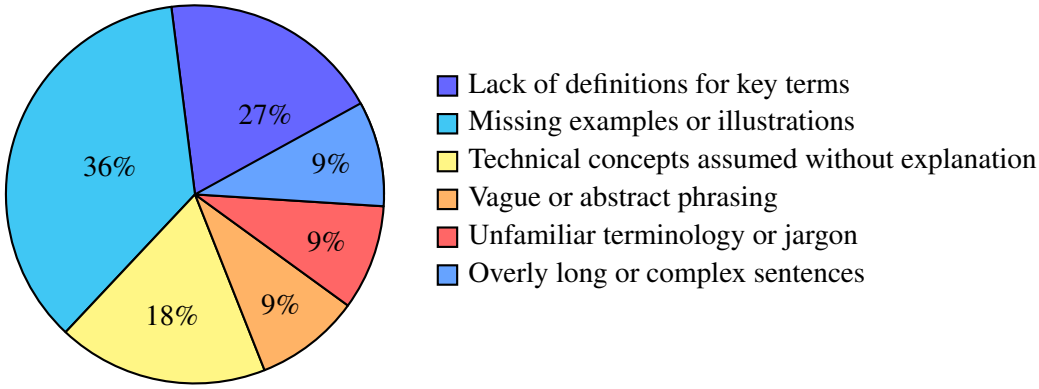

Figure 4.2: Aspects that made understanding the alert more difficult

In addition to these topics, some participants mentioned that the fact that the code was written in JavaScript was another factor that added difficulty to the task, as they were not familiar with the language and its specific features.

From these criticisms of the warning, feedback was gathered and suggestions were made for potential improvements to the alert that participants would like to see:

- Having access to more detailed error messages and links to official documentation.
- Providing a section of examples of possible fixes for the problem.
- Tests confirming that the solution worked and did not change the functioning of the code.

4.1.2.2 Information-Seeking Patterns and Questions

Finally, the other relevant aspect to analyze in order to understand the challenges experienced in solving the task was the nature of the questions and help requested by the participants during the interview. To this end, the two types of data were collected: 1) a history of searches and questions made to AI tools, and 2) a record of verbal questions asked aloud by the participants.

Table 4.3 presents the questions posed to external resources. The majority of assistance requested focused on clarifying the `eval()` function and identifying how to correct the vulnerability.

Tools Used	**Prompts**
ChatGPT	"What does eval function do in JavaScript" "If I wanted to make this code secure: val=eval(“val.” + kid); with JSON.parse() for instance how would I do it" “Codeql has detected a vulnerability in val=eval(“val.” + kid); Can you help in identifying the vulnerability and fixing it?” “This function has a cross-site scripting on val=eval(“val.” + kid); Fix it pls” “Cross-site scripting vulnerability due to a user-provided value. So I have this error on a query and it points to this line of code val=eval(“val.” + kid); What’s the problem here, and mainly why does my error happen?” “Hey, I have a function in JS that uses eval access to access deeply nested properties in JS objects as a string. What alternatives are there to eval?”
DeepSeek	"I’m analyzing the output of CodeQL report. It shows me this warning: Cross-site scripting vulnerability due to a user-provided. – insecure use of eval (simple name check) on my-database. What can I do to fix?" "How to avoid Cross-site scripting (XSS) Vulnerability in parameter variable with JavaScript" “In what follows, it is the extract of the code with the warning message: <code> What can do to fix?” “Could you please analyze the following JavaScript code: <code>”
Search Engines	"eval in javascript" "JS eval function" "JS eval xss" "eval secure in javascript" "Codeql insecure use of eval (simple name check)" "stringify js" "JavaScript object properties through string" "JS get object parameters" "JS access object property"

Table 4.3: Questions asked during the task

Meanwhile, Table 4.4 presents the questions asked aloud during the interview sessions, organized according to the specific stages of the task in which they were asked.

Phases	**Questions**
Understanding the warning	"Does it point me to the possible site of the vulnerability" (translated from PT to EN) "Do I need to click on something?" "So, from what I understand, the problem is, obviously, cross-site scripting, and it's due to the use of eval, right?" "What does this mean?" "What are the risks?"
Analyzing the general context	"But where is this [`kid` parameter] coming from?" "So what is this [the main function of the file]?" "Is this code from a library?" "What does the program do?" "Is this function supposed to concatenate `val` with `kid` here?" (translated from PT to EN) "Basically, it's going to add the `val`, that `val` string plus the `kid`, right? But then what is `val`?" (translated from PT to EN) "What exactly is `val`?" (translated from PT to EN) "Because I don't know what `val.kid` would be. What is the `val` in this case?"
Determine the action based on the line	"Should I replace the `eval()`?" "What should I do next?"
Fixing the issue	"In this case, would it be with `get()`" (translated from PT to EN) "If I put `val.kid` here, this won't work, right?" (translated from PT to EN) "How do I do validation in JavaScript?" (translated from PT to EN) "How could I write it without using `eval()`?" "How do I access properties in JavaScript?" (translated from PT to EN)

Table 4.4: Questions asked aloud by the participants

4.2 Discussion

This section presents the interpretation of the results, taking into account the initial objectives, in order to understand and analyze the implications of the conclusions drawn in this project.

The results obtained in this study provide relevant insights into the experience of developers when dealing with alerts produced by static analyzers.

Starting with the behavioral aspect, it can be concluded that although the participants did not consider the task to be particularly stressful, they did characterize it as a cognitively demanding task, requiring a moderate to high effort. Another factor to take into account was the request for external help to better understand technical aspects, such as how the `eval()` function works, and especially to correct the problem in the code, which reinforces this issue. It follows that these tasks apparently do not entail much emotional pressure or frustration, nonetheless they are a significant intellectual challenge for developers.

In relation to the main challenges, an important topic observed was the general difficulty in immediately understanding the cause of the warning and the code provided, due to the lack of information and clarity present in the warning.

Finally, it is also worth noting the need to use external resources, especially AI tools, to fix problems. This raises questions about the confidence and autonomy of developers in carrying out these tasks, which points to insecurity and poor understanding in interpreting the messages generated.

These conclusions point to the following implications:

- **The need to improve the tools' messages:** from the developers' observations and feedback, it is very clear that there is a need to make the alert messages more informative and useful, with the inclusion of context, definition of key terms and the cause of the problem and suggestions for fixes.

- **Educational support:** this topic is particularly relevant for less experienced developers, but the inclusion of guided explanations, with step-by-step reasoning examples explaining the problem and its solution, could make it easier to understand and fix it, as well as promoting good practices and increasing the quality of the code developed.

- **Build different warning messages**: based on the different chains of thought extracted in this study, there is a need to provide and build solutions and alert messages that address the different needs of developers.

4.3 Threats to validity

In this section, some aspects that can be considered a threat to this study are presented.

In the first place, there are internal threats that can be identified:

- **The presence of the interviewer:** despite the fact that the interviewer was impartial and did not provide any solutions or suggestions, their presence may have influenced the participant behavior, as well as some of the questions asked by the participants could have helped to give them some guidance.

- **Familiarity with the language used:** the fact that some participants were not familiar with JavaScript, whereas others were, could have affected their performance and difficulties.
- **Presentation of only one scenario:** the fact that only one relatively simple scenario (code repository and alert) was shown can limit the conclusions, since some of the participants could have previous knowledge the features of the language, which would help them solving the problem, while others may not, and if it had been used more than one that difference could have been balanced.

In second place, the external threats to the study are:

- **Sample size:** despite being a sample with diversity of knowledge and background, the fact that it is a small sample, 12 participants, may present some difficulties in generalizing the results.
- **Perception that participants have of the task and their performance:** performance assessment is subjective and the participants themselves evaluated it, this can add a little of bias to the assessment, since they might have different perceptions of what it means for a task to be successful, easy, etc.

Chapter 5

Developer Reasoning Improves LLM Explanations

This chapter presents a proof-of-concept solution that draws on the insights from Results and Discussion to explore how LLM-generated explanations can be improved. Based on the behavioral patterns identified in the user study, we designed a reasoning-based prompt that reflects how developers typically interpret and respond to SA warnings.

To explore its potential benefits, we compare this reasoning-based prompt to a baseline prompt that provides minimal guidance. Both prompts were given to `GPT-4.1` under identical conditions. The resulting outputs were evaluated based on a single criterion: determining which phases of the developers' chain of thought were addressed by the generated outputs. This criterion reflects the study's aim to assess alignment between AI-generated explanations and the reasoning patterns observed in developers (see Chapter 4).

In **Section 5.1**, we introduce the two prompt variants: a **developer-informed** prompt, structured around the observed chain of thought, and a **baseline** prompt with a minimal context. **Section 5.2** presents our exploratory evaluation, identifying which phases of the reasoning process are addressed by each output. Finally, **Section 5.3** reflects on the key takeaways and discusses how incorporating developer reasoning into prompt design can inform future tooling and explanation strategies.

5.1 Prompt Variants

To investigate whether explicitly incorporating a reasoning structure into the prompt improves the quality of generated explanations, we compared two distinct prompts: a **developer-informed** prompt, grounded in the chain of thought observed in our user study, and a **baseline prompt**, designed to reflect more typical, ad hoc interactions developers might have with LLMs.

5.1.1 Developer-Informed Prompt

Based on the behavioral patterns and information needs identified in Results and Discussion, we designed a prompt that mirrors how developers reason through static analysis alerts. Its structure was informed by insights from both research questions:

- **RQ1** underscored the importance of providing clear and detailed explanations about the cause of the alert. This includes a description of the use of `eval()`, an explanation of the broader program context and the role of the line that triggered the alert, as well as illustrative examples demonstrating how the vulnerability could be exploited. These elements correspond to the key reasoning steps identified in the developers' chain of thought (CoT1 in Table 4.1).
- **RQ2** revealed the types of questions developers most frequently asked and the points at which they sought external support. These included requests for safe and effective ways to fix the issue, as well as guidance on how to test and validate whether the vulnerability had been properly addressed.

The improved prompt is grounded in the step-by-step reasoning process identified as *CoT1* (in Table 4.1) and the questions posed by developers throughout the alert resolution stages (see Tables 4.3 and 4.4). Concretely, *CoT1* captures a five-step process observed across participants: (1) understanding the alert, (2) analyzing the code's context and purpose, (3) determining an appropriate action, (4) applying a code fix, and (5) validating that the issue is solved. To ensure comprehensive coverage, the main questions raised by participants during the study were reformulated as concrete actions and included in the prompt. An example of this transformation is the use of some of the questions from Table 4.4 of Understanding the warning, such as *"What does this mean?"* and *"What are the risks of this vulnerability?"*, which were turned into actions in the prompt: *describe the alert meaning and its risks*

Additionally, two design principles informed the prompt structure:

- **Start with relevant context:** Include the original warning message, the location of the issue, and the affected file to anchor the explanation.
- **Define a clear reasoning structure:** Request a step-by-step explanation aligned with the logical sequence observed in developer behaviour.

The final prompt used was:

> *A static analysis tool produced this alert: "Cross-site scripting due to a user-provided value from a static analysis tool", triggered in line 31 in the store.deep.js file. Provide a step-by-step reasoning to explain and address it. Your explanation must include the following steps: Understand the alert (describe what the alert means, the vulnerability and its potential impact), Analyze the code context (examine the surrounding code*

> *to understand why this alert was triggered and the purpose of the program), Determine the action to take (decide whether the alert is valid and what should be done), Fix the alert (suggest a fix for the alert) and Validate the solution (describe how to test if the issue is resolved, produce a automatic test and if the original functionality of the code is kept).*

5.1.2 Baseline Prompt

Before evaluating the reasoning-informed prompt, we first tested a baseline version intended to approximate how developers might interact with LLMs when seeking help with SA alerts. In informal settings, developers often ask for a fix or explanation in a direct and unstructured way, especially when they are unsure how to phrase a more specific request [Nam et al., 2024]. Our goal with this baseline prompt was to mimic this behavior: a straightforward request for assistance, without additional context or reasoning guidance. This allows us to evaluate how the model performs when left to interpret the task broadly and to assess whether embedding developer reasoning patterns into the prompt provides added value. Concretely, we used the following prompt:

> *A static analysis tool produced this alert: "Cross-site scripting due to a user-provided value from a static analysis tool", triggered in line 31 in the store.deep.js file. Provide a step-by-step reasoning to explain and address it.*

5.2 Reasoning-Aligned Assessment

To explore the benefits of prompting with a developer-informed reasoning structure, we performed a qualitative, side-by-side comparison of the outputs produced by each prompt. The two prompts were given to `GPT-4.1` under identical conditions using the CursorAI interface [CursorAI, 2025]. CursorAI was selected for its ability to provide context-aware assistance by analyzing the structure and semantics of the surrounding codebase – capabilities not available in standard LLM interfaces. Furthermore, we selected `GPT-4.1` as it was designed to outperform other models in its family (like GPT-4o and GPT-4.5), surpassing them in code-related tasks and instruction-following capabilities [Goose, 2025].

Rather than evaluating factual correctness or completeness in a formal sense, we analyzed each output based on whether it reflected five reasoning steps derived from our user study (see Prompt Variants). These steps correspond to key phases in how developers interpret and resolve static analysis alerts. Concretely: (1) Understand the warning, (2) Analyze the code context, (3) Determine the appropriate action, (4) Fix the problem, and (5) Validate the solution.

5.2.1 Results

The output generated using the baseline prompt and the developer-informed prompt is presented in Table 5.1. While Table 5.2 summarizes the reasoning steps reflected in both prompts.

Baseline Prompt	Informed Prompt
[STEP 1] **1. Understanding the Alert** • **Alert:** "Cross-site scripting due to a user-provided value" • **File:** `src/store.deep.js` • **Line 31:** `val = eval("val." + kid);`	**[STEP 1]** **1. Understand the Alert** • **Alert:** "Cross-site scripting due to a user-provided value from a static analysis tool", triggered in line 31 of the `store.deep.js` file. **What does this mean?** This alert is warning about a potential **Cross-Site Scripting (XSS)** vulnerability. XSS occurs when an attacker is able to inject malicious scripts into web pages viewed by other users. If user input is executed as code (e.g. via `eval`), an attacker could run arbitrary JavaScript in the context of the site, potentially stealing data, hijacking sessions, or performing actions on behalf of the user. **Potential Impact:** • Data theft (cookies, local storage, etc.) • Account/Session hijacking • Defacement or malicious actions on behalf of the user

[OC STEP]

2. Why is this a Problem?

`eval` Usage: The code uses `eval` to dynamically access a property path.

```
val = eval("val." + kid);
```

- **User Input:** The value of `kid` is derived from user input (the key passed to `store.get`), which can be manipulated by an attacker.
- **XSS Risk:** If an attacker can control `kid`, they could inject malicious JavaScript code, leading to Cross-Site Scripting (XSS).

[STEP 2]

2. Analyze the Code Context

Relevant Code (Lines 27–34):

```
else if (kid) {
  var val = _.parse(s);
  /*jshint evil:true */
  val = eval("val." + kid);
  s = _.stringify(val);
}
```

- The function is a deep getter for objects stored in a storage area.
- If a "kid" (child key) is provided, it parses the stored value and then uses `eval("val."+kid)` to access a nested property.
- The `kid` value is derived from the key, which can be user-controlled.

Why was the alert triggered?
The use of `eval` with a value (`kid`) that can be influenced by user input is dangerous. If an attacker can control `kid`, they can inject arbitrary code.

[OC STEP]

3. How Does the Vulnerability Work?

Suppose a user calls:

```
store.get('foo.is.not.quite');
```

- `kid` becomes `'is.not.quite'`.
- The code evaluates: `eval("val.is.not.quite")` (which is safe if the key is trusted).

But if an attacker calls:

```
store.get("foo'); alert('XSS');//"
    );
```

- `kid` becomes `'); alert("XSS")://'`.
- The code evaluates: `eval("val.');alert(XSS);//"`, which executes `alert("XSS")`.

[STEP 3]

3. Determine the Action to Take

Is the alert valid? Yes.

- Using `eval` on user-controlled input is a well-known security risk.
- Even if the intent is to access a nested property, there are safer alternatives.

What should be done?

- Remove the use of `eval`
- Replace it with a safe method to access nested properties (e.g., using property accessors or a loop).

[STEP 4]

4. How to Fix It?

Goal: Avoid using `eval` for property access.
Step-by-Step Solution:
1. Replace `eval` with Safe Property Access:
Instead of using `eval`, traverse the object properties using a loop.
2. Example Fix

```
//Replace this:
val = eval("val."+kid);

//With this:
var props = kid.split('.');
for (var i = 0; i < props.length;
    i++) {
    if (val == null) break;
    val = val[props[i]];
}
```

3. Why is This Safe? This approach only accesses properties by name, and does not execute arbitrary code.
It prevents injection of malicious code, mitigating XSS. Safer, avoids code execution.

[STEP 4]

Step 4. Fix the Alert

Safer Alternative: Instead of `eval("val."+kid)`, use a function to traverse the object properties safely.
Example fix:

```
else if (kid) {
  var val = _.parse(s);
  //Safely access nested
      properties
  var props = kid.split('.');
  for (var i = 0; i < props.length
      ; i++) {
    if (val == null) break;
    val = val[props[i]];
  }
  s = _.stringify(val);
}
```

This code splits the `kid` string by dots and accesses each property in turn, without using `eval`.

[OC STEP]

5. Summary (Flattened)

- **Step 1:** User input passed as `kid` – user controls property path.
- **Step 2:** `eval("val." + kid)` is called – risk of arbitrary code execution.
- **Step 3:** Replaced with property traversal – accesses only property names.
- **Step 4:** Fix applied – XSS risk mitigated.

[STEP 5]

Step 5. Validate the Solution

How to test if the issue is resolved:

- **Security:** Ensure that arbitrary code cannot be executed via the `key` to `kid` parameters.
- **Functionality:** Ensure that the function still correctly retrieves nested properties.

Automatic Test Example (using Jest or similar):

```
// Setup
store.set('foo', { is: { not: {
    quite: false }}});

// Test normal access
console.assert(store.get('foo.is.
    not.quite') === false, 'Should
    retrieve nested value');

// Test XSS attempt
let result = store.get('foo.is.not
    .constructor.constructor("alert
    (1)")()');
console.assert(result ===
    undefined, 'Should not execute
    injected code');
```

The first test checks that the original functionality is preserved.
The second test ensures that an XSS attempt does not execute code or throw errors.

[OC STEP]	**[STEP 6]**
6. Final Recommendation • Never use `eval` on user input. • Use property traversal for dynamic property access. • Test the fix to ensure functionality is preserved.	**Step 6. Summary (Flattened)** • **Step 1:** XSS risk due to `eval` on user input. • **Step 2:** `eval("val." + kid)` used to access nested property, `kid` is user-controlled. • **Step 3:** Alert is valid, must remove `eval`. • **Step 4:** Replace `eval` with safe property access (split and traverse object). • **Step 5:** Test for XSS and normal functionality with automatic tests

Table 5.1: Outcomes of the Baseline Prompt and the Developer-Informed Prompt. In steps without the tag [STEP X] do not correspond to the step in CoT1

The **baseline prompt** elicited four reasoning steps, but its structure did not align with the stages defined in the chain of thought of the user study. Instead of mirroring the reasoning process observed in developers, the model's output focused primarily on the alert itself. Concretely, it followed this sequence:

Step 1 – *Understanding the Alert* (corresponding to *Understanding the Alert* in CoT1): It only provided base information on the warning (alert message and file location), but did not explain the vulnerability or its risks;

Step 2 – *Why is this a Problem?*: Only presented a brief explanation of the issue and the vulnerability, and offered no context about the overall program;

Step 3 – *How Does the Vulnerability Work?*: Presentation of potential exploitation scenarios for the vulnerability, but it does not clearly state whether it is valid or what action should be taken;

Step 4 – *How to Fix It?* (corresponding to *Fix the Alert* in CoT1): suggestion for replacing the function and implementation of the fix.

While these steps formed a coherent flow, this chain focuses on addressing the alert in isolation instead of considering its context within the broader codebase.

In contrast, our **developer-informed prompt**, which was explicitly designed to reflect the reasoning patterns observed in developers, yielded a response that closely followed the 5 reasoning steps empirically extracted from developers' behavior, showing a closer alignment with real-world problem-solving approaches. The output produced by the developer-informed prompt is well-structured and delineates each reasoning step, briefly describing the actions to be taken. Concretely:

Step 1 – *Understanding the Alert*: The model answers the question "What does this alert message mean?" by presenting the warning, identifying the vulnerable code location, and explaining the nature and potential impact of the issue.

Step 2 – *Analyze the Context*: A code snippet surrounding the line that caused the alert is presented, with an explanation of its purpose and a special focus on the wrapper in line 31. The model also explains why the alert was triggered, effectively answering the questions "What does the program do?" and "Why was the alert triggered?".

Step 3 – *Determine The Action*: It is ascertained whether the alert could be a false positive and explains why it is valid. The model then proposes replacing the use of `eval()`. In addition, this step answers questions such as "should I fix this?" and "what should I do?"

Step 4 – *Fix the Problem*: A brief reference is made to the solution to correct the problem, finding a safe way to traverse the properties of the object in question, `val`. Then follows the implementation of the fix. In this way, it answers questions asked during the phase of fixing the problem, "how do I do validation in JavaScript?" and "how could I write it without using `eval()`?".

Step 5 – *Validate the Solution*: The model explains how to validate the suggested fix. An example of an automatic test is presented to check that the alert has been corrected and that the original operation of the program is maintained.

Prompt	Understand the warning	Analyze the context	Determine the action	Fix the problem	Validate the solution
Baseline Prompt	Y	N	N	Y	N
Developer-Informed Prompt	Y	Y	Y	Y	Y

Table 5.2: Steps reported in the results of the prompts tested in `GPT-4.1`

The comparison between the two prompts reveals significant differences in both the structure and depth of the generated explanations. Although the baseline prompt followed a plausible reasoning sequence, its focus remained restricted to the alert itself. In contrast, the developer-informed prompt presents a structured reasoning process aligned with the CoT1 in Table 4.1. Each step includes the actions required and offers detailed explanations that reflect the actual difficulties developers reported when dealing with static analyzer alerts.

Ultimately, Table 5.2 shows that the improved prompt, which embeds the five-step reasoning chain, consistently led to outputs that explicitly address each reasoning step. Whereas, the baseline prompt—designed to reflect common ad-hoc developer queries—often skipped key steps such as analyzing context or validating the solution. This suggests that prompting with an explicit reasoning structure can steer the model toward more interpretable and systematic outputs, even if we do not evaluate correctness per se.

5.3 Discussion and Takeaways

Our results highlight a key insight: guiding large language models to reason through static analysis alerts in a manner consistent with how developers actually approach such tasks leads to significantly better outcomes.

The developer-informed prompt led to more complete, structured, and actionable responses, unlike the baseline prompt, which skipped important stages such as assessing context.

Beyond improved content quality, structuring prompts around human reasoning steps can potentially enhance other important aspects of developer support:

- **Usability**: Presenting the steps alongside some interactive questions allows developers to skip steps they find irrelevant, providing flexibility and a more streamlined user experience.
- **Accessibility**: Since all essential information to understand and fix the alert is embedded within the alert, developers would not need to rely on external resources, reducing cognitive overhead and context switching.
- **Educational**: Exposing the reasoning behind each step can support a deeper understanding of vulnerabilities, secure coding practices and code quality.

These observations suggest that prompting strategies grounded in real-world developer reasoning may contribute not only to more effective model outputs but also to more usable, accessible, and educational interactions, particularly when integrated into development tools.

Chapter 6

Conclusion and Future Work

This chapter presents the conclusions of the study and outlines directions for future work. It begins by summarizing the study's key contributions in light of the original objectives, followed by suggestions for how the insights gained can inform and guide future research and development.

6.1 Conclusion

This dissertation achieved its initial objectives by analyzing the chain of thought followed by developers when engaging with the outputs of static analyzers, the behaviors they adopted while attempting to understand and solve the reports presented, and the key deadlocks they encountered while interpreting and addressing these alerts.

The user study successfully answered the two RQs defined at the beginning of the project. Regarding RQ1 (*behavioral patterns* and *chain of thought*), the study revealed that developers have difficulties in fully understanding the causes of the alerts, which made them unsure about how to fix them. Despite the task requiring moderate to high intellectual effort (average mental demand of 5.1 out of 10 and an effort of 4.9 out of 10), participants did not find it stressful (3.4/10). A common chain of thought characterizing the reasoning process followed by participants when addressing the report was extracted. This chain of thought was followed by 67% of participants, while the remaining 33% skipped one or more steps. It consisted of five steps: (1) Understand the Alert, (2) Analyze the Code Context, (3) Determine the Action to Take, (4) Implement the Fix, and (5) Test the Solution. Concerning RQ2 (*cognitive challenges*), the main challenges observed were that the alert was broad and lacked sufficient contextual information, such as a technical explanation about the function that triggered the alert and its associated risks. It also did not provide an illustrative example of the problem or concrete suggestions for how to fix the issue. Consequently, participants sought external assistance, both through search engines and AI tools, and voiced questions aloud during the task, indicating gaps in their understanding and the need for more helpful and actionable guidance. This help was requested to better understand the technical terms and to find a solution to the problem.

Thus, these findings offer valuable insights into the real-world behavior and actions of developers when interacting with static analysis tools and opportunities for tool improvement. The main contributions of this study are based on the interpretation and comprehension phase, where developers consistently require more assistance, especially in understanding the reason why the alert was triggered and how it relates to the code context. Once this is achieved, resolving the problem becomes more straightforward, even though developers are often uncertain about their fixes and seek help to complete them.

To summarize, these results highlight the need for clearer and more context-aware alert messages to support developers in making informed decisions. Providing explanations alongside examples of safer alternatives for correcting the code could significantly streamline the fixing process, increase productivity, and enhance the confidence with which developers address security issues.

6.2 Future Work

Based on the observations and conclusions drawn both in the study and the proof of concept developed, this project could be further explored with the purpose of applying this knowledge.

A promising direction for future work is the integration of LLMs into static analysis tools to enhance their usability and educational value. One possible application is to design interactive support features inspired by the general chain of thought extracted (CoT1) and the proof of concept. For each step of the reasoning chain, a set of follow-up questions could be created, adapted from those originally asked by participants, both in searches and aloud. These questions could be transformed into clickable buttons within the alert, providing access to explanations in an interactive and user-friendly way. Implementing this would require testing prompts across different models and evaluating them to identify the most effective one. Once the LLM is chosen, it could be integrated into a static analyzer tool, for example, CodeQL. The final step would involve the validation of the tool through a user study, similar to the methodology used in this work.

Beyond the improvement of technical implementation (creating better messages), another interesting direction to explore is the leveraging of LLMs for educational purposes. By integrating structured results and interactive chains of thought into static analyzers, as described previously, LLMs could serve not only as automated and augmented reasoning engines but also as assistants, guiding users through complex analysis, explaining concepts and promoting security practices. It is therefore important to further investigate the role that this integration can play in educational terms, especially the impact it has on less experienced developers. For example, this could be researched through a user study divided into two scenarios. In the first, participants would be presented with an alert generated by one of the current static analysis tools. In the second, they would receive the alert accompanied by a structured chain of thought explanation generated by an LLM. This would enable a comparison between the two interactions and user experiences, allowing for the evaluation of the two approaches.

References

[Alahmadi et al., 2022] Alahmadi, B. A., Axon, L., and Martinovic, I. (2022). 99% false positives: A qualitative study of SOC analysts' perspectives on security alarms. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2783–2800, Boston, MA. USENIX Association.

[AlOmar and Mkaouer, 2024] AlOmar, E. A. and Mkaouer, M. W. (2024). Cultivating software quality improvement in the classroom:an experience with chatgpt. pages 1–10.

[Ayala and Garcia, 2023] Ayala, J. and Garcia, J. (2023). An empirical study on workflows and security policies in popular github repositories. In *2023 IEEE/ACM 1st International Workshop on Software Vulnerability (SVM)*, pages 6–9.

[Braz and Bacchelli, 2022] Braz, L. and Bacchelli, A. (2022). Software security during modern code review: the developer's perspective. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2022, page 810–821, New York, NY, USA. Association for Computing Machinery.

[Center, 2020] Center, N. A. R. (2020). Nasa tlx. https://humansystems.arc.nasa.gov/groups/tlx/. Accessed: 2025-06-15.

[Chapman et al., 2024] Chapman, P. J., Rubio-González, C., and Thakur, A. V. (2024). Interleaving static analysis and llm prompting. In *Proceedings of the 13th ACM SIGPLAN International Workshop on the State Of the Art in Program Analysis*, SOAP 2024, page 9–17, New York, NY, USA. Association for Computing Machinery.

[Christakis and Bird, 2016] Christakis, M. and Bird, C. (2016). What developers want and need from program analysis: an empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, page 332–343, New York, NY, USA. Association for Computing Machinery.

[CodeQL, 2025] CodeQL (2025). About data flow. https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/. Accessed: 2025-06-23.

[CursorAI, 2025] CursorAI (2025). Cursorai. https://www.cursor.com/. Accessed: 2025-06-24.

[DataDog, 2025] DataDog (2025). What is static analysis and how does it work? https://www.datadoghq.com/knowledge-center/static-analysis/. Accessed: 2025-06-19.

[Do et al., 2022] Do, L. N. Q., Wright, J. R., and Ali, K. (2022). Why do software developers use static analysis tools? a user-centered study of developer needs and motivations. *IEEE Transactions on Software Engineering*, 48(3):835–847.

[GitHub, 2025a] GitHub (2025a). Codeql. https://codeql.github.com/. Accessed: 2025-06-07.

[GitHub, 2025b] GitHub (2025b). Osv. https://osv.dev/vulnerability/GHSA-w5hq-hm5m-4548. Accessed: 2025-06-07.

[Goose, 2025] Goose, B. (2025). How gpt-4.1 compares to gpt-4o. https://medium.com/@leucopsis/how-gpt-4-1-compares-to-gpt-4o-5e7d9a52d113. Accessed: 2025-06-25.

[Hao et al., 2023] Hao, Y., Chen, W., Zhou, Z., and Cui, W. (2023). Ev: Prompting large language models to perform static analysis by pseudo-code execution and verification.

[Haonan Li, 2023] Haonan Li, Yizhuo Zhai, Y. H. Z. Q. (2023). Assisting static analysis with large language models: A chatgpt experiment. *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1–4.

[Johnson et al., 2013] Johnson, B., Song, Y., Murphy-Hill, E., and Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681.

[Khare et al., 2025] Khare, A., Dutta, S., Li, Z., Solko-Breslin, A., Alur, R., and Naik, M. (2025). Understanding the effectiveness of large language models in detecting security vulnerabilities. In *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 103–114.

[Li et al., 2024] Li, H., Hao, Y., Zhai, Y., and Qian, Z. (2024). Enhancing static analysis for practical bug detection: An llm-integrated approach. *Proc. ACM Program. Lang.*, 8(OOPSLA1).

[Li et al., 2025] Li, Y., Yao, P., Yu, K., Wang, C., Ye, Y., Li, S., Luo, M., Liu, Y., and Ren, K. (2025). Understanding industry perspectives of static application security testing (sast) evaluation. *Proc. ACM Softw. Eng.*, 2(FSE).

[Livshits and Lam, 2005] Livshits, V. B. and Lam, M. S. (2005). Finding security vulnerabilities in java applications with static analysis. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, page 18, USA. USENIX Association.

[Mamede, 2025] Mamede, C. (2025). Interpretable vulnerability detection reports. *Under Review*.

[Mao et al., 2025] Mao, Q., Li, Z., Hu, X., Liu, K., Xia, X., and Sun, J. (2025). Towards explainable vulnerability detection with large language models.

[Mohajer et al., 2024] Mohajer, M. M., Aleithan, R., Harzevili, N. S., Wei, M., Belle, A. B., Pham, H. V., and Wang, S. (2024). Effectiveness of chatgpt for static analysis: How far are we? In *Proceedings of the 1st ACM International Conference on AI-Powered Software*, AIware 2024, page 151–160, New York, NY, USA. Association for Computing Machinery.

[Mohajer et al., 2023] Mohajer, M. M., Aleithan, R., Shiri, N., Harzevili, Wei, M., Belle, A. B., Pham, H. V., and Wang, S. (2023). Skipanalyzer: A tool for static code analysis with large language models. pages 1–10.

[Nachtigall et al., 2019] Nachtigall, M., Nguyen Quang Do, L., and Bodden, E. (2019). Explaining static analysis - a perspective. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 29–32.

[Nachtigall et al., 2022] Nachtigall, M., Schlichtig, M., and Bodden, E. (2022). A large-scale study of usability criteria addressed by static analysis tools. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2022, page 532–543, New York, NY, USA. Association for Computing Machinery.

[Nam et al., 2024] Nam, D., Macvean, A., Hellendoorn, V., Vasilescu, B., and Myers, B. (2024). Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA. Association for Computing Machinery.

[Nations, 2025] Nations, U. (2025). The 17 goals sustainable development. https://sdgs.un.org/goals. Accessed: 2025-06-25.

[OSV, 2025] OSV (2025). Ghsa-w5hq-hm5m-4548. https://osv.dev/vulnerability/GHSA-w5hq-hm5m-4548. Accessed: 2025-06-24.

[OWASP, 2025a] OWASP (2025a). Owasp top ten. https://owasp.org/www-project-top-ten/. Accessed: 2025-06-23.

[OWASP, 2025b] OWASP (2025b). What is static analysis and how does it work? https://owasp.org/www-community/controls/Static_Code_Analysis. Accessed: 2025-06-19.

[Roest et al., 2024] Roest, L., Keuning, H., and Jeuring, J. (2024). Next-step hint generation for introductory programming using large language models. In *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, page 144–153, New York, NY, USA. Association for Computing Machinery.

[Sheese et al., 2024] Sheese, B., Liffiton, M., Savelka, J., and Denny, P. (2024). Patterns of student help-seeking when using a large language model-powered programming assistant. In *Proceedings of the 26th Australasian Computing Education Conference*, ACE '24, page 49–57, New York, NY, USA. Association for Computing Machinery.

[Tooley, 2024] Tooley, E. (2024). Appsec is harder than you think. here's how ai can help. https://github.blog/security/application-security/appsec-is-harder-than-you-think-heres-how-ai-can-help/?ocid=eml_pg465768_gdc_comm_gh. Accessed: 2025-06-04.

[Venkatesh et al., 2024] Venkatesh, A. P. S., Sabu, S., Mir, A. M., Reis, S., and Bodden, E. (2024). The emergence of large language models in static analysis: A first look through micro-benchmarks. In *Proceedings of the 2024 IEEE/ACM First International Conference on AI Foundation Models and Software Engineering*, FORGE '24, page 35–39, New York, NY, USA. Association for Computing Machinery.

[Yang et al., 2020] Yang, X., Yu, Z., Wang, J., and Menzies, T. (2020). Understanding static code warnings: An incremental ai approach. *www.elsevier.com/locate/eswa*.

Appendix A

Appendix

The following appendices include complementary resources that supported the study, namely the questionnaires for the first and last phases.

A.1 Pre-Screening Form

This form was used to register for the study, as well as to analyze the participants' background and skills in the area of security and static code analysis. The answers to this questionnaire made it possible to define profiles of the participants.

LLMs and Static Analysis in Security | Pre-Screening

Purpose of the Study

This pre-screening questionnaire is part of a Master's Dissertation exploring what developers feel about static analysis warnings and how Large Language Models (LLMs) can complement static analysis tools in a security education context. We are particularly interested in understanding developers' reasoning when identifying and fixing security-related issues.

What Will Happen Next

- After you complete this questionnaire, we will review your background information.
- Selected participants will be contacted to schedule an interview session in which they will be asked to work with security-related static analysis reports.

Thinking Aloud Protocol & Recoding

If you are invited to participate in the **next stage** of the study, you may be asked to use a "thinking aloud" approach while working on research scenarios. This means we will ask you to verbalize your thoughts as you interpret security warnings and decide how to address them. In order to capture your reasoning process, we will **record your voice (and potentially your screen activity)** during these tasks. These recordings will be used **solely for internal analysis** and will remain **strictly confidential**. They will **not** be shared outside the research team or published in any form. The goal is to understand your decision-making process and better map out your chain of reasoning when working with security warnings.

* Indica uma pergunta obrigatória

Consent

Voluntary Participation
Your participation in this study is entirely voluntary. You may decline to answer any question or withdraw from the study at any point, without explanation or penalty.

Confidentiality & Data Collection
We will only collect your email address for the purpose of contacting you about this and any subsequent phases of the study. We strongly encourage you **not** to share any other personally identifiable information in your responses or during the interview. If such information is shared inadvertently, rest assured that it will be removed or anonymized before any data analysis, and it will not be shared outside the research team. Final results will be reported only in an anonymized or aggregated form, and no personally identifiable information will be published.

Recording
With your permission, we will record the interview (audio and screen). Audio recording is especially important because we will ask you to "think aloud," describing your thoughts and reasoning processes in real-time. These recordings are used solely for research analysis, remain secure, and will not be shared outside the research team.

Duration
The interview is expected to last approximately 45 minutes. You are free to take breaks or ask for clarification at any time.
Consent Confirmation
By agreeing to the terms above, you acknowledge that you understand the purpose and procedures of this study and consent to participate under the conditions described. If you have any questions or concerns before or during the study, please discuss them with the research team.

1. I have read and accepted the conditions presented *

   Marcar apenas uma oval.

   ◯ Yes
   ◯ No

2. Email *

   _____

Background & Security Experience

3. **Development Experience.** How many years of experience do you have in software development (including academic, personal, or professional projects)?

   Marcar apenas uma oval.

   - ◯ Less than 2 years
   - ◯ Between 2 and 4 years
   - ◯ Between 4 and 6 years
   - ◯ More than 6 years

4. **Security Exposure.** Have you ever worked on security-related aspects of software development (e.g., secure coding, fixing vulnerabilities, threat modeling, code reviews with a security focus)?

   Marcar apenas uma oval.

   - ◯ Yes
   - ◯ No
   - ◯ Not directly, but I've encountered some security concerns

5. **Frequency.** How often do you encounter or resolve security-related issues (e.g., software vulnerabilities) in your work?

   Marcar apenas uma oval.

   - ◯ Never
   - ◯ Rarely
   - ◯ Sometimes
   - ◯ Often
   - ◯ Very often

6. **Familiarity with vulnerabilities.** How familiar are you with common security vulnerabilities such as cross-site scripting (XSS), SQL injection, or CSRF?

   Marcar apenas uma oval.

   - ◯ Not at all familiar
   - ◯ Slightly familiar
   - ◯ Moderately familiar
   - ◯ Very familiar
   - ◯ Extremely familiar

Confidence with Security Issues

7. **Reasoning-oriented Confidence.** How confident are you in **understanding** what caused a security warning and how it could be exploited?

   Marcar apenas uma oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very Confident

8. **Action-oriented Confidence.** How confident are you in **handling** security warnings in code (e.g., reviewing, triaging, or fixing them)?

   Marcar apenas uma oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very confident

9. **Factors for ignoring warnings.** What factors make you more likely to ignore a security warning?

Marcar tudo o que for aplicável.

- [ ] Alert seems low impact
- [ ] Alert appears to be a false positive
- [ ] Deadline pressure
- [ ] Unsure how to fix
- [ ] Tool lacks explanation
- [ ] Unclear or overly technical message
- [ ] Assumed someone else is responsible
- [ ] Outra: _____

10. **First reaction to a warning.** When you see a security warning in code, what do you usually do first?

Marcar tudo o que for aplicável.

- [ ] Try to understand what it means
- [ ] Search online or look at documentation
- [ ] Check if it's a false positive
- [ ] Ask a teammate for help
- [ ] Skip or suppress it
- [ ] Try a fix and test
- [ ] It depends on the context
- [ ] Outra: _____

Use of SAST

11. **Static analysis tools used.** Have you used static analysis before? If so, which ones?

Marcar tudo o que for aplicável.

- [ ] Semgrep
- [ ] CodeQL
- [ ] SonarQube
- [ ] Bandit
- [ ] ESLint (with security rules)
- [ ] Fortify
- [ ] I'm not sure
- [ ] I haven't used any
- [ ] Outra: _____

SAST Experience & Trust

🎉 **If you answered "I'm not sure" or "I haven't used any" to the previous question ("Have you used static analysis before?"), please SKIP this section!**

12. **Encountering false positives.** Have you ever encountered a false positive from a static analysis tool for a *security issue* (e.g., a vulnerability that turned out to be harmless)?

Marcar apenas uma oval.

- ◯ Yes
- ◯ No
- ◯ I'm not sure

13. **Response to false positives.** If so, how did you respond?

Marcar tudo o que for aplicável.

- [ ] Investigated further
- [ ] Ignored it
- [ ] Suppressed it
- [ ] Fixed it anyway
- [ ] Reported it
- [ ] Outra: _____

14. **Trust in SAST alerts.** How much do you trust security alerts generated by static analysis tools?

Marcar apenas uma oval.

	1	2	3	4	5	
Not	◯	◯	◯	◯	◯	Very much

15. **Helpful warning experience.** Have you ever found a static analysis warning helpful in understanding or fixing a security issue?

Marcar apenas uma oval.

- ◯ Yes
- ◯ No
- ◯ I'm not sure

16. **External Resources.** When you're unsure about a security warning from a static analysis tool, what external resources do you usually consult to validate or understand it better?

Marcar tudo o que for aplicável.

- ☐ Official documentation of the tool
- ☐ CWE or CVE databases
- ☐ Stack Overflow or forums
- ☐ GitHub issues or pull requests
- ☐ Security blogs or cheat sheets
- ☐ AI tools (e.g., ChatGPT, GitHub Copilot)
- ☐ Ask a colleague or mentor
- ☐ I usually don't validate with external tools
- ☐ Outra: _____

17. **Reason for helpfulness.** If so, what made it helpful or not helpful?

Thank you!

Thank you for your interest and willingness to take part in this research. Your insights will help us improve security education and the application of LLMs in static analysis contexts.

If you have any questions about the study or your rights as a participant, please contact the research team at up202007210@edu.fe.up.pt.

A.2 Post-Interview Form

This form was used to collect information about the participants' experience in the interviews and complement the data observed during the task.

LLMs and SAST in Security Education (Post-Interview)

Purpose of the Study

This pre-screening questionnaire is part of a Master's Dissertation exploring what developers feel about static analysis warnings and how Large Language Models (LLMs) can complement static analysis tools in a security education context. We are particularly interested in understanding developers' reasoning when identifying and fixing security-related issues.

Thinking Aloud Protocol & Recoding

If you are invited to participate in the **next stage** of the study, you may be asked to use a "thinking aloud" approach while working on research scenarios. This means we will ask you to verbalize your thoughts as you interpret security warnings and decide how to address them. In order to capture your reasoning process, we will **record your voice (and potentially your screen activity)** during these tasks. These recordings will be used **solely for internal analysis** and will remain **strictly confidential**. They will **not** be shared outside the research team or published in any form. The goal is to understand your decision-making process and better map out your chain of reasoning when working with security warnings.

Privacy

Thank you for taking the time to participate in our survey. Your responses are important to our research.
While some questions are open-ended to allow you to express your thoughts freely, please refrain from providing any personal or personally identifiable information about yourself or others, unless specifically requested.

* Indica uma pergunta obrigatória

Consent

Voluntary Participation
Your participation in this study is entirely voluntary. You may decline to answer any question or withdraw from the study at any point, without explanation or penalty.

Confidentiality & Data Collection
We will only collect your email address for the purpose of contacting you about this and any subsequent phases of the study. We strongly encourage you **not** to share any other personally identifiable information in your responses or during the interview. If such information is shared inadvertently, rest assured that it will be removed or anonymized before any data analysis, and it will not be shared outside the research team. Final results will be reported only in an anonymized or aggregated form, and no personally identifiable information will be published.

Recording
With your permission, we will record the interview (audio and screen). Audio recording is especially important because we will ask you to "think aloud," describing your thoughts and reasoning processes in real-time. These recordings are used solely for research analysis, remain secure, and will not be shared outside the research team.

Duration
The interview is expected to last approximately 30 minutes. You are free to take breaks or ask for clarification at any time.

Consent Confirmation
By agreeing to the terms above, you acknowledge that you understand the purpose and procedures of this study and consent to participate under the conditions described. If you have any questions or concerns before or during the study, please discuss them with the research team.

1. Participant ID *

_____

2. I have read and accepted the conditions presented *

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

Protocol

In this exercise, you'll be working with a real security alert generated by a static analysis tool called CodeQL.

You'll use GitHub Codespaces, which provides a pre-configured development environment with all necessary tools and code already set up. We will also provide GitHub credentials for access.Before you begin:
✅ Confirm that you can access GitHub Codespaces.
✅ Confirm that you are sharing your screen with the interviewer.
✅ Keep this questionnaire open during the exercise.

🧩 **Your Task**
You'll be shown a security alert produced by CodeQL. Your goal is to:
🧠 Understand what the alert is trying to say
🛠️ Investigate the code and attempt to fix the issue
🌐 Feel free to use any external resources you'd normally use.

This is an open-ended task. We're interested in how you think, where you get stuck, and what helps you move forward.

Go to the next page, when the task is completed!

Post Study Survey

The following section is very important to this study. Please, answer all questions to the best of your ability.

3. **In your own words, what security issue is CodeQL warning you about in this alert?** Specify the lines or code pattern that triggered the alert, and why is that pattern potentially dangerous.

4. **How clear was the alert/report to you?**
(Scale: 1 = Very Unclear; 10 = Very Clear)

Marcar apenas uma oval.

	1	2	3	4	5	6	7	8	9	10	
I did	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	I knew exactly what it meant

5. **Explain what made the report clear or unclear to you.**

_____

_____

_____

_____

_____

6. **How did you feel reading the report?**

Marcar apenas uma oval.

- ◯ Frustrated
- ◯ Confident
- ◯ Unsure
- ◯ Overwhelmed
- ◯ Outra: _____

7. **If you found a similar security alert in your daily work, what would you do?**

Marcar apenas uma oval.

- ◯ Prioritize
- ◯ Ignore
- ◯ Take immediate action
- ◯ Outra: _____

8. **Explain why you would make such decision.**

9. **Did you attempt to fix the issue?**

Marcar apenas uma oval.

- ◯ Yes
- ◯ No

10. **Explain why.**

11. **What tools (if any) did you use to help you during the task?**

Marcar tudo o que for aplicável.

- ☐ Google
- ☐ Stack Overflow
- ☐ ChatGPT or another AI assistant
- ☐ Documentation
- ☐ Outra: _____

12. **Which of the following made understanding the alert/vulnerability most difficult?**

Marcar apenas uma oval.

- ◯ Unfamiliar terminology or jargon
- ◯ Vague or abstract phrasing
- ◯ Lack of definitions for key terms
- ◯ Overly long or complex sentences
- ◯ Missing examples or illustrations
- ◯ Technical concepts assumed without explanation
- ◯ Outra: _____

13. **Which of the following challenges did you still encounter even after using external tools?**

Marcar tudo o que for aplicável.

- ☐ Understanding the root cause behind the alert
- ☐ Mapping the alert back to the exact code lines
- ☐ Interpreting the severity or impact of the vulnerability
- ☐ Knowing which remediation steps to take first
- ☐ Locating up-to-date documentation on the vulnerability
- ☐ Integrating suggested fixes into the existing codebase
- ☐ Verifying that your fix fully addresses the issue

14. **Which changes (if any) would you perform to simplify the task?**

_____

_____

_____

_____

_____

NASA TLX

In this study, we use the NASA-TLX questionnaire to evaluate your perceived workload while performing the assigned task.

12. **Which of the following made understanding the alert/vulnerability most difficult?**

Marcar apenas uma oval.

- ◯ Unfamiliar terminology or jargon
- ◯ Vague or abstract phrasing
- ◯ Lack of definitions for key terms
- ◯ Overly long or complex sentences
- ◯ Missing examples or illustrations
- ◯ Technical concepts assumed without explanation
- ◯ Outra: _____

13. **Which of the following challenges did you still encounter even after using external tools?**

Marcar tudo o que for aplicável.

- ☐ Understanding the root cause behind the alert
- ☐ Mapping the alert back to the exact code lines
- ☐ Interpreting the severity or impact of the vulnerability
- ☐ Knowing which remediation steps to take first
- ☐ Locating up-to-date documentation on the vulnerability
- ☐ Integrating suggested fixes into the existing codebase
- ☐ Verifying that your fix fully addresses the issue

14. **Which changes (if any) would you perform to simplify the task?**

_____

_____

_____

_____

_____

NASA TLX

In this study, we use the NASA-TLX questionnaire to evaluate your perceived workload while performing the assigned task.

15. **How mentally demanding was the task?**

Marcar apenas uma oval.

	1	2	3	4	5	6	7	8	9	10	
Very	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Very High

16. **How hurried or rushed was the pace of the task?**

Marcar apenas uma oval.

	1	2	3	4	5	6	7	8	9	10	
Very	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Very High

17. **How successful were you in accomplishing what you were asked to do?**

Marcar apenas uma oval.

	1	2	3	4	5	6	7	8	9	10	
Perf	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Failure

18. **How hard did you have to work to accomplish your level of performance?**

Marcar apenas uma oval.

	1	2	3	4	5	6	7	8	9	10	
Very	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Very High

19. **How insecure, discouraged, irritated, stressed, and annoyed were you?**

Marcar apenas uma oval.

	1	2	3	4	5	6	7	8	9	10	
Very	◯	◯	◯	◯	◯	◯	◯	◯	◯	◯	Very High

Thank you!

Thank you for your interest and participation in the study. Your contribution is invaluable to the insights and conclusions we aim to draw from this project.