UNIVERSITY OF SALERNO

Department of Computer Science

Master Degree in Computer Science
Curriculum of Software Engineering and IT Management

THESIS

CRANE - Code Review AI Network Engine

SUPERVISOR

Prof. **Fabio Palomba**

Dr. **Gilberto Recupito**

Dr. **Antonio Della Porta**

Dr. **Giammaria Giordano**

University of Salerno

Prof. **Giuseppe Cascavilla**

Prof. **Damian A. Tamburri**

Jheronimus Academy of Data Science

CANDIDATE

Luca Morelli

Mat.: 0522501590

Academic Year 2024/2025

This thesis was carried out in

"It is not our part to master all the tides of the world, but to do what is in us for the juice of those years wherein we are set, uprooting the evil in the fields that we know, so that those who live after may have clean earth to till. What weather they shall have is not ours to rule."

J.R.R. Tolkien

"Non tocca a noi dominare tutte le maree del mondo; ma fare il possibile per la salvezza degli anni nei quali viviamo, sradicando il male dai campi che conosciamo, affinché chi verrà dopo trovi una terra più pulita da coltivare. Il tempo che avranno non dipende da noi."

J.R.R. Tolkien

Abstract

Secure Code Review (SCR) is increasingly vital for identifying vulnerabilities early in the software development lifecycle. However, it remains a time-consuming and resource-intensive task, especially in large and fast-paced projects. To address this, many studies have proposed the adoption of AI-based solutions to support or automate parts of the review process. While LLMs can accelerate secure code review and reduce developer workload, relying on them without proper oversight may introduce subtle yet critical vulnerabilities, thus posing significant risks to software security. This thesis proposes CRANE—a collaborative multi-agent framework that simulates code review through configurable agents, each representing either a human or a Large Language Model (LLM), and supporting both fully automated and human-in-the-loop pipelines.

Developed using the DSR methodology, CRANE addresses key limitations of LLM-based code reviews identified in prior studies. It was evaluated on 366 change requests from Gerrit, using 2,522 Semgrep rules to assess vulnerabilities. Evaluation focused on vulnerability count, severity, and semantic similarity.

The best-performing configuration of CRANE achieved an 85.91% reduction in vulnerabilities compared to the human-written baseline, while maintaining a cosine similarity of 0.691. Personality-based agents and vulnerability information improved results, whereas increasing the number of rounds or LLMs led to diminishing returns and higher severity, underscoring the need for tailored configurations.

CRANE is a valuable multi-agent LLM framework for both software practitioners and educators, offering insight generation, snippet suggestions, and collaborative review capabilities. Future work may test CRANE on additional languages, integrate advanced RAG techniques, and explore its effectiveness in human-AI workflows.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Early Detection of Vulnerabilities

A **vulnerability** is defined as "*a flaw or weakness in a system's design, implementation, or operation that could be exploited to violate the system's security policy*" [1]. Addressing these vulnerabilities early in the Software Development Life Cycle (SDLC) is critical, as remediation costs tend to increase significantly when issues are discovered in later stages [2].

To address this risk, many organizations are adopting a "shift-left" approach, embedding security practices earlier in the SDLC [3]. However, early integration alone is not always sufficient. Developers must be motivated and aware of the importance of **identifying vulnerabilities** [4], and even then, effectively locating and addressing security issues within an organization remains a significant challenge [5]. Additionally, working in isolation has been shown to increase the likelihood of introducing preventable security flaws [6]. One practice that addresses both the timing and collaboration challenges of vulnerability detection is the Code Review Process.

1.2 Code Review Process and Security Vulnerabilities

The **Code Review Process** is widely recognized as a best practice in software engineering [7]. It is valued not only for reducing software defects but also for improving overall code quality [8, 9, 10].

However, despite its benefits, the code review process presents several well-known challenges [11]. First, conducting thorough code reviews can be **costly** in terms of time and resources. Ensuring high-quality reviews often requires significant investment, especially in large or fast-paced projects, costs not always feasible for smaller development teams.

Finally, effective code reviews depend heavily on the **expertise of reviewers** or may require training and support to ensure that they are familiar with the Code Review Process and tools. In many cases, it is not feasible to always have domain experts available, which limits the consistency and reliability of the review process. These challenges make Code Review a practice that, while beneficial, is not always easily replicable or viable in all development environments.

Beyond quality assurance, prior research has shown that Code Review plays a crucial role in **identifying and addressing security vulnerabilities** early in the development cycle [12, 13], contributing positively to secure software development practices [14, 15, 16].

However, adopting effective security-focused review practices requires significant domain knowledge, which can be difficult to acquire and even harder to motivate among developers [17, 18]. This knowledge barrier poses a challenge to the consistent application of secure coding principles during code reviews.

1.3 Checklists & Instructions for Secure Code Review

Building on this, Braz et al. [19] conducted an experiment involving 150 software practitioners—each with a minimum of three years of experience—to investigate the impact of explicit instructions and checklists on **vulnerability detection** during Code Review. The study focused on Java code containing two common vulnerabilities: CWE-209 (Information Exposure Through an Error Message) and CWE-327 (Use of a

Broken or Risky Cryptographic Algorithm).

Participants were divided into four groups: No Instructions (NI), Security Instructions (SI), Security Checklist (SC), and Tailored Security Checklist (TC). The results showed that **explicitly instructing participants to focus on security** increased the likelihood of identifying vulnerabilities by a factor of eight. Interestingly, the use of checklists—whether generic or tailored—did not lead to a significant improvement in vulnerability detection rates.

This finding somewhat contrasts with the qualitative feedback. While some participants found the checklists helpful, others criticized them as vague or lacking in specificity. A commonly cited reason for their limited effectiveness was the reviewers' lack of prior security knowledge.

Ultimately, Braz et al. [19] concluded that while security-focused instructions significantly aid vulnerability detection, checklists alone may not offer the same benefits. In this context, a **"less is more" approach** may be more effective—emphasizing clear, concise security guidance over complex or generalized solutions like checklists to improve the code review process.

1.4 Developer Perspectives on Secure Code Review

Braz et al. [20] conducted a mixed-methods study involving interviews with 10 professional developers and a survey of 182 practitioners to explore secure code review from a **developer's perspective**. Their findings reveal that most developers do not actively prioritize security during code review—unless specifically prompted to do so. When asked general questions about what they focus on during code review, only 9 out of 182 participants explicitly mentioned security, despite 81% of them acknowledging that identifying vulnerabilities is part of their responsibility. This discrepancy suggests that security is often not top-of-mind during typical code reviews or is considered less frequently than developers claim.

Multiple factors contribute to this gap. Some reviewers justified that security was not their primary responsibility or assumed that internal code did not need to meet the same security standards. While organizations expect security to be addressed during code review, many developers believe they lack the necessary institutional

support. Notably, two-thirds of participants reported that their **organizations do not offer dedicated security training**. In fact, 44 respondents cited lack of knowledge and training as a major obstacle.

Ultimately, Braz et al. [20] recommend a shift in organizational mindset: companies should not only expect secure practices from their developers but also actively support them. This could involve investing in structured security education, incentivizing secure behaviors, and involving security experts directly in the code review process.

1.5 AI-Generated Code and Security Concerns

While Large Language Models (LLMs) have shown great potential in accelerating code generation [21], their use in security-critical contexts raises significant concerns. Relying solely on AI to recover or enhance code quality can be suboptimal and risky.

Negri et al. [22] conducted a systematic analysis revealing that LLMs frequently **introduce well-known security vulnerabilities** during code generation. Many of these issues align with the MITRE CWE Top 25 Most Dangerous Software Weaknesses, demonstrating that the problem is not limited to obscure or edge-case bugs. These vulnerabilities have been observed even in mature and widely adopted programming languages such as Python and Java.

This highlights a key limitation: LLMs—at their current state—cannot be safely deployed in fully autonomous development pipelines. Human oversight remains essential to guide inputs, configure parameters, and supervise overall system behavior to ensure both code quality and security.

🔍 On LLMs & Programming Practices

Although LLMs often produce syntactically correct and functionally plausible code, they **lack the contextual awareness and judgment** needed to enforce secure programming practices consistently.

1.6 Proposed Solution: CRANE

To address the challenges CRANE (Code Review AI Network Engine) has been developed. CRANE's primary goal is to **assist developers by generating high-quality actionable suggestions and refined code snippets** in response to change requests.

CRANE features a modular, multi-agent architecture, in which each agent can be instantiated as either a Large Language Model (LLM) or a human participant. This design supports human-in-the-loop configurations, allowing developers to directly intervene at specific stages of the pipeline. This not only enhances transparency, trust, and oversight but also reduces the costs needed to execute a Code Review and the need for deep expertise on security while making it more scalable.

Each LLM agent is assigned a **unique demographic persona**, a Myers-Briggs Type Indicator (MBTI) **personality type** suited to its role [23], and a **specialization** in a specific software engineering domain (e.g., cybersecurity, testing, or quality assurance). These diverse attributes aim to simulate realistic human behavior and introduce a range of perspectives into the review process.

Unlike traditional single-agent or rule-based approaches, CRANE's collaborative multi-agent structure allows for inter-agent dialogue, enabling LLMs with complementary expertise to discuss, critique, and refine suggestions. This cooperative mechanism contributes to the generation of more secure and contextually relevant code or suggestions. In doing so, CRANE addresses two critical shortcomings issues in current software development and code review workflows:

- Checklists that provide only informations instead of actionable security insights;
- Insecure code produced by isolated LLMs.

Ultimately, CRANE offers a structured, scalable, and adaptive approach to conducting code reviews that are both **technically sound** and **security-conscious**, while supporting **human-in-the-loop approaches**.

1.7 Methodology Summary & Key Results

CRANE was developed following the Design Science Research (DSR) methodology, addressing key challenges identified in recent literature on secure code review and AI-assisted development [19, 20, 22]. To evaluate its effectiveness, we first conducted a **sequential ablation study** of CRANE's fully automated pipeline to identify the best-performing configuration. We then performed a comparative analysis using a dataset of **real-world change requests** sourced from the Gerrit code review platform, curated by Mukadam et al. [24].

The evaluation focused on comparing CRANE's outputs with actual developer-written code, specifically with regard to the presence of security vulnerabilities. This analysis aimed to assess CRANE's capability to autonomously generate more secure code. To provide additional context, we also measured vulnerability severity using Semgrep and semantic similarity using cosine similarity scores.

Results indicate that the best-performing configuration (C3) reduced vulnerabilities by 81.95% compared to human-written code, with mostly of remaining issues classified as medium severity. Furthermore, C3 maintained a high degree of semantic alignment with the original code, achieving an average cosine similarity score of 0.691, suggesting that CRANE's output closely resembles the intent and structure of developer-written code.

1.8 Contributions, Audience & Impact

This study offers four key contributions:

- A **multi-agent LLM-based system** capable of simulating collaborative Code Review Processes;
- A **flexible human-in-the-loop architecture**, allowing developers to actively participate as agents within the review workflow;
- A curated **dataset** documenting the outputs of seven different CRANE configurations provided via Zenodo [25];

- A **sequential ablation study** evaluating the impact of specific framework parameters—such as MBTI persona type, number of reviewers, and iteration count—on the semantic alignment and security of the generated code.

The implications of this work span multiple domains:

- **Software Engineering Research**: CRANE contributes to studies in multi-agent systems, AI-for-SE (AI4SE), and human-in-the-loop development methodologies;
- **AI and NLP Research**: The system demonstrates the use of advanced prompt engineering, persona modeling, and agent interactions with LLMs;
- **Software Development Teams**: CRANE can be integrated or adapted into existing code review pipelines to enhance security and collaboration;
- **Tool Builders and Platform Integrators (e.g., GitHub, Gerrit):** CRANE's architecture could inform the development of intelligent, multi-agent Code Review recommendation tools;
- **Educators**: The simulation of peer feedback makes CRANE a promising tool for teaching secure and effective code review practices in academic or training settings.

CHAPTER 2

Related Works

Secure Code Review (SCR) practices are gaining increasing importance as organizations seek to address **security concerns early in the Software Development Life Cycle** (SDLC). While AI models have shown potential as supportive companions in this process, existing frameworks for AI-assisted Code Review remain largely unexplored.

To provide a comprehensive understanding of the current landscape, this chapter offers an in-depth overview of the state-of-the-art in SCR practices. It begins with an examination of traditional, human-centric approaches, followed by an exploration of emerging strategies that integrate AI-human collaboration. The discussion then shifts to recent developments in AI-generated code, with a particular focus on the associated security vulnerabilities.

The chapter concludes by identifying critical gaps in the current body of research—specifically, the lack of solutions that effectively combine automation with security awareness and human oversight. These gaps highlight the need for a novel framework, motivating the introduction of **CRANE**, which aims to address these shortcomings through a flexible, collaborative, and security-oriented approach.

2.1 Secure Code Review Process

2.1.1 From Code Weaknesses to Security Issues

Charoenwet et al. [26] conducted an in-depth empirical study on two major open-source projects—OpenSSL and PHP—to investigate how code weaknesses—often precursors to security vulnerabilities—are addressed during the code review process. Their analysis focused on CWE-699, which includes 40 categories of coding weaknesses often associated with security vulnerabilities.

The study found that coding weaknesses were raised 21 to 33.5 times more frequently than explicit vulnerabilities during reviews, but they were not always effectively addressed. The authors categorized the outcomes as follows:

- **Fix attempted:** In 39%–41% of cases, developers actively attempted to fix the identified issue.
- **Acknowledged:** Between 30%–36% of weaknesses were acknowledged but not immediately resolved. Among these, 10%–18% were fixed in later commits, while 18%–20% remained unresolved due to **disagreements about the appropriate solution**.
- **Dismissed:** About 14%–26% were dismissed after discussion, often because they were deemed false positives or acceptable by design.
- **Unresponded:** Approximately 3%–9% received no response and were effectively abandoned.

Overall, the results indicate that many code weaknesses are frequently discussed during code reviews. However, the depth and consistency of attention vary. Interestingly, weaknesses linked to well-known vulnerabilities, such as buffer overflows or resource management errors, were among the **least frequently discussed**. This suggests that some security-relevant issues may still be underrepresented in current review practices.

2.1.2 Developers' Attitudes and Challenges in Secure Code Review

Braz et al. [20] conducted a mixed-method study combining interviews with 10 professional developers and a survey involving 182 practitioners to explore how developers perceive and handle security during code reviews.

Most developers did not spontaneously mention security as a primary concern; only 9 out of 182 respondents explicitly cited it when asked about what they focus on during reviews. Instead, non-functional qualities such as maintainability or readability were more frequently referenced. However, when explicitly prompted about security, up to **111 respondents** reported that they **always consider vulnerabilities** during code reviews. This suggests that security may be considered **less frequently in practice** than developers self-report, possibly due to **misplaced assumptions**.

For example:

- 7 interviewees believed that ensuring security was the responsibility of another team (e.g., dedicated security teams).
- Others assumed that internal code—especially code not intended for production—did not require security checks.

Despite this, **81%** of respondents claimed they have the autonomy to decide what issues to inspect during reviews, and the same proportion acknowledged that they **personally feel responsible** for identifying vulnerabilities.

When considering organizational support:

- 149 respondents think companies should do **more to promote secure practices**.
- Two-thirds reported that their companies **do not provide security training**.
- Only **24%** felt recognized by their organizations or project leads for conducting security-focused reviews.

Concerning structural support:

- Only **47%** stated their organization has a dedicated security team.
- A mere **12%** said their code changes are reviewed by security experts.

Finally, several practical challenges were identified:

- **44 respondents** highlighted a lack of security knowledge and training as the most frequent issue.
- 9 mentioned that vulnerabilities are often difficult to detect.
- 6 pointed to insufficient time and resources, especially given heavy workloads.
- Additional obstacles included reviewing long code snippets, limited knowledge of system architecture, interaction with other components, and the use of third-party libraries.

2.1.3 Limitations of Static Analysis Tools for Secure Code Review

Lenarduzzi et al. [27] conducted an extensive comparison of six widely-used Static Analysis Tools (SATs) for Java projects. The study evaluated each tool in terms of issue detection, agreement with other tools, and precision (i.e., ratio of true positives to all detected issues) across 47 open-source Java projects, with results shown in Table 2.1.

- **Better Code Hub:** This tool focuses on structure, organization, modifiability, and comprehensibility. It reported **27,888 issues**, but showed **low agreement** with other tools. Its precision was only **29%**, indicating that nearly two-thirds of its recommendations were false alarms.
- **CheckStyle:** An open-source tool designed to enforce Java coding standards. It reported the **highest number of issues** (**9,686,813**) but achieved the **highest precision** of **86%**, largely due to its focus on syntactic issues (e.g., indentation checks). However, many of these do not directly affect code security or behavior, suggesting that CheckStyle should be complemented with other tools for security-related assessments.
- **Coverity Scan:** This tool detects vulnerabilities and defects grouped by category. It reported **7,431 issues** with **low agreement** with other tools and a precision of **37%**, again meaning that the majority of issues could be false positives.

- **FindBugs:** Uses bug-pattern recognition to identify common coding defects. It found **33,704 issues** and had the **highest overlap** with PMD (9.378%). Precision was **57%**, suggesting that nearly half the identified issues may not be valid concerns.
- **PMD:** Applies a set of rules to detect problems such as unused variables and unnecessary object creation. It identified **3,380,493 issues** and had relatively low agreement with other tools (except FindBugs). Its precision stood at **52%**.
- **SonarQube:** Evaluates compliance with rule sets for multiple programming languages. It found **418,433 issues**, had low agreement with others, and the **lowest precision** (**18%**). This is attributed to the tool's high sensitivity and broad detection scope, which increases false positives.

Overall, the study illustrates that current SATs often show poor inter-tool agreement and high false positive rates. While some tools, such as CheckStyle, achieve high precision, they tend to focus on superficial syntax rather than security-relevant issues.

Table 2.1: Comparison of Static Analysis Tools (SATs) for Java Projects

Tool	Number of Issues	Agreement	Precision
Better Code Hub	27,888	Low	29%
CheckStyle	9,686,813	Low	86%
Coverity Scan	7,431	Low	37%
FindBugs	33,704	Low	57%
PMD	3,380,493	Low	52%
SonarQube	418,433	Low	18%

2.2 Human-AI Collaboration

Recent studies have highlighted the growing role of AI in software engineering workflows [28, 29], particularly in enhancing productivity and supporting developers through intelligent code assistance.

Yadav et al. [28] emphasized that AI models are not limited to generating code from scratch. Instead, they can support a wide range of software engineering tasks, including:

- **Translation between programming languages**, facilitating cross-language collaboration and interoperability;
- **Code review assistance**, including the identification of security vulnerabilities and code smells;
- **Code reorganization**, aimed at improving code readability or performance;
- **Code completion**, offering suggestions during the development process;
- **Natural language processing for documentation and sprint planning**, enhancing both communication and project management;
- **Integration into CI/CD pipelines**, such as generating unit tests or synthetic data.

Fan et al. [29] further explored the impact of AI in pair programming scenarios. They found that AI partners can mitigate common limitations of traditional (human-human) pair programming, such as skill disparities between partners and challenges in collaborative planning. AI can offer:

- **Immediate feedback** and on-demand assistance;
- **Flexibility in interaction**, allowing developers to engage with the AI partner at any time;
- **Reduced programming anxiety** and **increased motivation**, due to the absence of social judgment;
- **Performance improvements**, with AI-assisted groups outperforming both individuals and human-human teams on programming tasks.

These findings suggest that AI assistants can partially replicate the collaborative benefits of human partnerships while also contributing positively to the learning experience.

2.2.1 Challenges and Ethical Considerations

Despite these advantages, both Yadav et al. [28] and Fan et al. [29] caution against overreliance on AI systems. Several challenges and risks remain:

- **Dependence on AI**, which may hinder the development of problem-solving and critical thinking skills;
- **Reduced effort and engagement**, as easy access to solutions could discourage in-depth learning and practice;
- **Bias in AI models**, since training data may reflect human biases that lead to discriminatory outputs;
- **Transparency and accountability issues**, as integrating LLMs into development workflows can obscure decision-making and weaken responsible AI practices.

In conclusion, Yadav et al. [28] recommend a balanced approach: while AI tools can automate repetitive or low-level tasks, human developers should **retain control over high-level design** and creative problem-solving. Human-in-the-loop approaches remain essential to ensuring both the reliability and ethical use of AI in software development.

2.3 AI code generation

2.3.1 Security of AI-Generated Code

Large Language Models (LLMs) have seen widespread adoption in code generation tasks [21, 22, 30, 31]. This capability, commonly referred to as Natural Language to Code (NL2Code), has attracted significant interest in both research and industry. However, the use of LLMs for generating code also raises critical concerns related to security.

As highlighted by Basic et al. [31], current LLMs often lack awareness of secure coding principles, resulting in code that frequently contains vulnerabilities. Among the most prevalent issues are **injection vulnerabilities**, which are consistently found

in AI-generated code across various domains. The security level of generated code, however, appears to be influenced by the target programming language [22]. For instance, AI-generated Python code is often more secure than code generated in C or Verilog. These differences are commonly attributed to the volume and quality of training data available for each language.

Negri et al. [22] further emphasized that LLMs are capable of introducing well-known vulnerabilities, including those listed in the MITRE Top 25 Common Weakness Enumerations (CWEs). In particular, CWE-787 (Out-of-bounds Write) and CWE-089 (SQL Injection) were found to be among the most frequently introduced issues, highlighting the **recurring nature of certain vulnerability patterns** in AI-generated code.

2.3.2 Security Analysis Across Programming Languages

As previously highlighted, Negri et al. [22] examined the security of AI-generated code across four different programming languages. Their findings revealed notable differences in vulnerability prevalence depending on the language:

- **Python:** AI-generated Python code tends to be more secure than code generated in languages like C. However, Pearce et al. [32] found that 38.25% of Python code produced by AI models contained at least one vulnerability from the MITRE Top 25 (2021), indicating that the security risks remain substantial.

- **C/C++:** Code generated in C or C++ often contains a higher number of vulnerabilities. Approximately 50% of AI-generated C programs were found to be insecure. The comparative security of AI versus human-generated C code remains unclear. Sandoval et al. [33] noted that results vary depending on the metrics used, with AI-assisted developers sometimes outperforming models like Codex and vice versa. Meanwhile, Asare et al. [34] observed that AI models tend to make fewer security-related mistakes in some contexts.

- **Java:** AI models frequently produce insecure Java code, proposing **twice as much SSTuB** (Stupid Single Bug) as correct code. Wu et al. [35] reported that Codex resolved only 20.4% of vulnerabilities in Java code. Additionally, Tony

et al. [36] highlighted, while working with Java, that AI might not be at all optimized for generating cryptographically secure code, with an accuracy significantly lower on cryptographic tasks than what AI is advertised to have on regular code.

- **Verilog:** AI models are generally less effective when generating Verilog code, often producing non-functional or incorrect implementations. This difficulty is exacerbated in longer prompts. Unlike software-oriented languages, evaluating Verilog code for vulnerabilities is challenging, as there is no established Top 25 CWE list for hardware, as noted by Negri et al. [22].

2.3.3 Comparing AI-Generated and Human-Written Java Code

An insightful comparison between human-written and AI-generated Java code was conducted by Hamer et al. [37]. Their study evaluated code produced by ChatGPT against snippets sourced from Stack Overflow, revealing several noteworthy findings regarding security vulnerabilities.

The AI-generated code contained 248 vulnerabilities, compared to 302 identified in the Stack Overflow code—indicating a **20% reduction in vulnerabilities** for the AI-generated solutions. Furthermore, ChatGPT's code introduced only 19 distinct types of CWEs, whereas Stack Overflow examples exhibited 22.

The most commonly observed CWEs in both sets were CWE-327 (Use of a Broken or Risky Cryptographic Algorithm) and CWE-328 (Use of Weak Hash). Notably, ChatGPT outperformed Stack Overflow in avoiding CWE-335 (Incorrect Usage of Seeds in Pseudo-Random Number Generators) and CWE-835 (Loop with Unreachable Exit Condition). However, Stack Overflow had a better record in avoiding CWE-798 (Use of Hard-coded Credentials).

In total, three vulnerabilities listed in the MITRE Top 25 CWEs (2023) were identified in the analyzed code: CWE-078 (OS Command Injection), CWE-476 (NULL Pointer Dereference), and CWE-798.

Ultimately, Hamer et al. [37] emphasize a critical point: regardless of whether code is generated by humans or AI, it should **never be used blindly**.

2.3.4 Prompt Engineering for Safer Code

Prompt engineering techniques can play a crucial role in minimizing the vulnerabilities introduced by large language models (LLMs), as well as in reducing false positives during vulnerability detection tasks.

Jiang et al. [21] observed that LLMs, such as ChatGPT, often produce a significant number of false positives when used to detect vulnerabilities. However, they found that **structuring prompts** more effectively—such as asking the model to summarize functions, decompose tasks step-by-step, and iteratively refine results—can **improve detection accuracy** and reduce the rate of false positives.

Similarly, Noever et al. [38] reported promising results when using GPT-4 for vulnerability detection. Their methodology required the model not only to identify potential issues but also to **justify its reasoning** and correct any hallucinatory or incorrect responses. This approach contributed to a notably low false positive rate, demonstrating the importance of enforcing structured and reflective responses.

Interestingly, Wang et al. [39] found that providing insecure code explanations in code repair tasks did not significantly improve the model's performance. Their results suggest that **insecure code explanations**, while potentially useful for human understanding, **do not provide substantial assistance** for LLMs in automated vulnerability repair scenarios.

2.4 Research Gaps

Despite the growing body of work on human-AI collaboration in software development [21, 22, 30, 31], the integration of human-in-the-loop mechanisms within collaborative, multi-agent Large Language Model (LLM) systems remains largely underexplored. Most existing studies focus on either fully automated code generation or isolated human evaluations, overlooking the potential of dynamic, iterative interaction between humans and multiple AI agents during the vulnerability detection and remediation process.

This thesis seeks to address that gap by proposing a flexible framework that supports both fully automated pipelines and human-in-the-loop configurations. Such

a system enables not only **automation for experienced users** but also provides a **guided environment for developers with less security expertise**.

In fully automated setups, predefined demographic persona patterns must be specified, requiring the user to possess a contextual understanding of the task to achieve. Conversely, the human-in-the-loop mode introduces opportunities for human reasoning, contextual decision-making, and collaborative debugging—while still benefiting from the breadth of knowledge and consistency that LLMs can offer.

This dual-mode system responds directly to the challenges highlighted in the literature: the scarcity of developer expertise in security-related tasks, the difficulty of identifying vulnerabilities within isolated workflows, and the general need for earlier, collaborative Secure Code Reviews. By embedding LLMs into a collaborative workflow rather than using them in isolation, this research aims to bridge the usability-expertise gap and **enhance the effectiveness of vulnerability management practices**.

CHAPTER 3

Research Method

This section describes the research methodology adopted to design, develop, and evaluate CRANE. We followed the Design Science Research Method [40] to ensure a structured approach from problem identification to solution validation. Our aim was to investigate whether a multi-agent LLM-based system could enhance the Code Review Process by simulating diverse perspectives.

3.1 Overview of the Research Design

The study follows the Design Science Research (DSR) methodology [40] to develop and evaluate CRANE, a multi-agent LLM system that simulates code review processes while trying to reduce code vulnerabilities. The research design comprises three main phases:

- **Problem Identification and Solution Design:** Based on existing literature highlighting challenges in secure code review practices [20], we designed CRANE to simulate diverse reviewer personas using LLM agents.
- **Implementation:** CRANE was implemented with a modular architecture supporting human-in-the-loop or fully automated configurations. Agents are as-

signed demographic and personality profiles [23] to foster varied perspectives during code review simulations.

- **Evaluation:** we conducted a comparative analysis using a real-world dataset of Gerrit change requests [24], assessing CRANE's performance through vulnerability counts to human-written code.

3.2 Applied Research Method

3.2.1 Overview on the Design Science Research Method

During the development of CRANE, we adopted the Design Science Research Methodology (DSRM), which is grounded in the Design Science paradigm. This approach focuses on extending human and organizational capabilities through the creation of innovative artifacts [40]. Within this paradigm, knowledge and understanding of a problem domain are advanced through the iterative design, construction, and evaluation of purposeful IT artifacts.

In line with this approach and the specific challenges CRANE addresses, we also applied a mixed-method research strategy:

- a **qualitative approach** was used to interpret and apply vulnerability severity rankings in code snippet evaluations.
- a **quantitative approach** was employed to analyze numerical outcomes such as the number of vulnerabilities and semantic similarity scores (based on cosine similarity).

This combination allowed for both interpretive insight and empirical validation.

3.2.2 Alignment between DSRM & CRANE development

The motivation for adopting the Design Science Research Method (DSRM) stems from two key insights in the literature. First, Braz et al. [20] highlight that, although organizations express a strong interest in software security, few concrete actions are

taken to incorporate security considerations during code reviews. Second, recent studies [39, 22] warn against the uncritical use of Large Language Models (LLMs) in code generation, due to their potential to introduce subtle yet dangerous vulnerabilities.

These concerns underscore the need for systems that not only assist developers during the Code Review process but also enhance the security of code generated by LLMs. In response, this study aims to design and validate CRANE—a problem-driven, multi-agent LLM framework specifically tailored to improve the security outcomes of code review activities.

This research goal is well aligned with the principles of DSRM: CRANE was designed to directly address clearly identified real-world problems, and its effectiveness can be empirically validated using robust evaluation metrics, such as the number and severity of code vulnerabilities and semantic similarity with human-written code snippets.

3.2.3 DSRM Main Phases

As outlined by Hevner et al. [40], the Design Science Research Methodology (DSRM) comprises several iterative phases that guide the construction and evaluation of artifacts:

- **Problem Identification:** The process begins with identifying a relevant and challenging real-world or literature-based problem that justifies the need for a novel solution.
- **Research Question Definition:** Based on the problem context and preliminary analysis, a set of targeted Research Questions (RQs) are formulated to guide the study.
- **Design & Development:** An artifact is designed and implemented with the intent to address the RQs and resolve the identified problem.
- **Demonstration:** The artifact is demonstrated in a relevant context to show its applicability and use.

- **Evaluation:** The artifact's effectiveness is rigorously evaluated using appropriate empirical or analytical methods, typically aligned with the RQs.
- **Communication of Results:** The final phase involves documenting and disseminating the findings to both academic and practitioner communities.

In the context of this study, all the aforementioned steps were explicitly applied in the development and evaluation of CRANE.

Each of these phases is further detailed in Section 3.3.1, where their practical implementation in the CRANE framework is described.

3.2.4 Strengths of the DSRM

Although widely adopted, the Design Science Research Method (DSRM) [40] presents both strengths and limitations when applied to artifact development in computing and information systems. The main strengths of the DSRM include:

- **Problem-solving orientation:** DSRM is inherently focused on addressing real-world problems, which ensures that developed artifacts are practically relevant to both industry and practitioners.
- **Balance of theory and practice:** By integrating rigorous theoretical grounding with practical application, DSRM ensures that artifacts are both academically sound and practically effective.
- **Focus on innovation:** The methodology encourages the creation of novel artifacts rather than the critique of existing ones—an advantage especially relevant in fast-evolving technological domains.
- **Iterative and flexible structure:** Its iterative nature allows researchers to continuously refine artifacts in response to evolving requirements and insights. Further implementation details are provided in Section 3.3.2.
- **Traceability of design choices:** DSRM supports a transparent and structured research process, which enhances the traceability and reproducibility of design decisions.

3.2.5 Limitations & Mitigation Strategies

Despite its strengths, several challenges are associated with applying DSRM. However, these can be mitigated through thoughtful design and contextual adaptation:

- **Limited generalizability:** Artifacts developed through DSRM may be highly tailored to specific problems, limiting their broader applicability. In CRANE, this limitation is addressed by leveraging a modular architecture and generic LLM components, allowing adaptability through prompt engineering and configuration adjustments.
- **Risk of insufficient theoretical grounding:** DSRM projects may be perceived as overly applied, with inadequate engagement with academic theory. To counter this, CRANE's design is explicitly informed by literature across multiple domains, including: (i) optimal number of reviewers [41, 42]; (ii) the use of MBTI personality traits in software engineering [23, 43, 44]; and (iv) known risks of vulnerability generation in LLMs [31, 22].
- **High interdisciplinary knowledge demand:** Effective application of DSRM requires expertise across several domains. To address this, an extensive literature review was conducted covering Secure Code Review Practices, Secure Code Generation, and SATs.
- **Lack of familiarity or acceptance:** DSRM may not be universally recognized in fields that prioritize empirical or theoretical paradigms. To increase transparency and acceptance, this thesis explicitly details each methodological step and its rationale, as elaborated in Section 3.3.1.

3.3 Artifact Development

CRANE (Code Review AI Network Engine) is designed to support developers throughout the Code Review Process by providing actionable feedback that can be utilized by either human developers or AI-based code generators. In the subsequent

section, the application of methodological steps to develop CRANE and the more informations about it will be provided

3.3.1 CRANE's Methodological Steps

In line with the Design Science Research (DSR) methodology, the development of CRANE followed a structured process composed of the following steps:

- **Problem Identification:** A thorough review of the literature was conducted to identify key challenges in secure AI-driven code generation [31, 22], the role of MBTI personality types in software engineering [23, 43, 44], and the human-limitations during code review processes [20]. These insights informed the design requirements of CRANE.
- **Research Question Formulation:** Based on the identified problems and the envisioned artifact, a set of research questions (RQs) was defined to guide the project's direction and evaluation criteria.
- **Design & Development:** CRANE was designed and implemented as a modular, multi-agent system capable of integrating both LLMs and human participants. Further architectural details are presented in Section 3.3.2.
- **Demonstration:** CRANE was tested using real-world code review (CR) tasks extracted from the Gerrit repository [24]. The system was provided with CR tasks and the associated code snippets, allowing us to observe how the collaborative multi-agent LLMs processed and contributed to the review workflow.
- **Evaluation:** The system was evaluated through both quantitative and qualitative analyses to assess its effectiveness in addressing the defined RQs. Details regarding the evaluation process are discussed in Section 4.2.4.
- **Communication of Results:** The results of the study, including datasets, prompts, and configuration details, have been made available on Zenodo [25, 45], and are also reported in detail in Chapter 5.

The iterative element of the DSRM was addressed by progressively improving the model in response to newly discovered insights, as well as through successive refinement of the prompts used within CRANE. Initially, baseline prompts were evaluated based on the quality and relevance of the generated responses. These prompts were then iteratively adjusted to better align with the intended task outcomes, improving clarity, precision, and alignment with each agent's role and context. All finalized prompts are publicly available on Zenodo [45] for transparency and reproducibility.

3.3.2 Modular Structure & Architecture

To ensure modularity, scalability, and reusability, CRANE follows a modular architecture with a waterfall execution flow: the output of each phase is passed as input to the next, as shown in Figure 3.1.

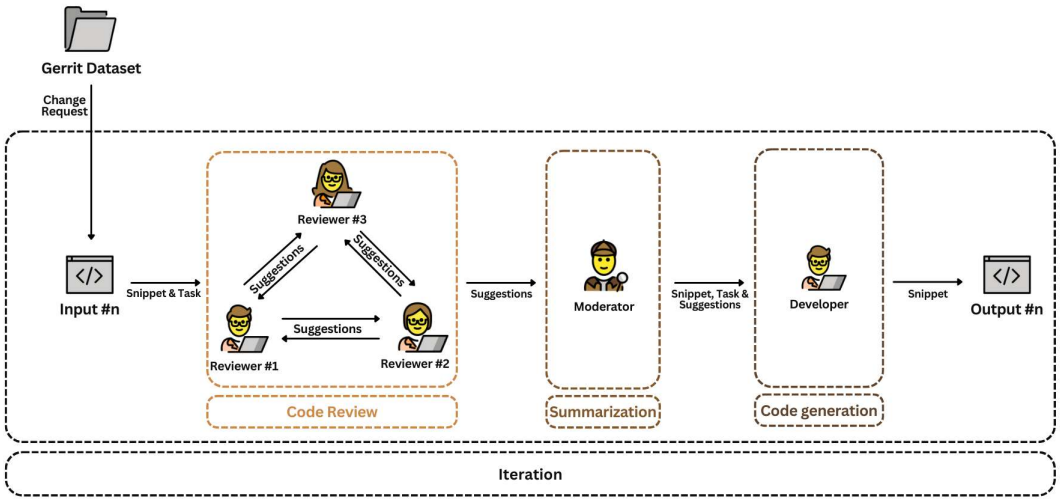

Figure 3.1: CRANE Framework

The system is structured into three main phases:

- **Code Review Phase:** Each agent, referred to as a Reviewer, contributes a fixed number of suggestions. The first round, called the **Individual Review**, involves agents analyzing the task and corresponding code snippet independently, without access to others' feedback. Starting from the second round—termed the

Collaborative Review—agents are granted access to a shared conversational history, allowing them to read and build upon the suggestions provided by their peers. Additionally, while a response was mandatory during the **Individual Review**, in the **Collaborative Review** phase reviewers could choose not to respond; in such cases, their suggestion field was left blank.

- **Summarization Phase:** The list of collected suggestions is passed to a dedicated **Moderator** agent. The Moderator reorganizes and synthesizes the suggestions into a concise and coherent response.
- **Code Generation Phase:** The refined suggestions, along with the original task and code snippet, is then provided to the **Developer**, whose role is to generate a revised version of the snippet based on the accumulated suggestions.

At the end of each iteration, the output consists of an updated code snippet paired with the original review task. This output can then serve as the input for the next iteration, allowing multi-step refinement of the code.

3.3.3 Agent Interaction Model

CRANE employs a message-passing strategy for inter-agent communication within each iteration. This approach is supported by a **conversational shared history**, which functions as a dynamic memory shared among Reviewers. It allows agents to access prior suggestions made during the same iteration, fostering coherence and collaborative refinement.

To enable multi-iteration workflows and long-term memory, CRANE integrates a Retrieval-Augmented Generation (RAG) mechanism through a **shared-memory strategy**. After each iteration, the Moderator's summarized output is stored in the RAG module. In subsequent iterations, this information becomes accessible to all reviewers, ensuring continuity and informed decision-making across the review process.

3.3.4 Agent Configuration & Human-in-the-Loop Integration

To ensure modularity and support flexible customization, each CRANE agent is configured through a dedicated JSON file. This configuration defines key attributes such as MBTI personality type, assigned role (e.g., Reviewer, Moderator, or Developer) and domain specialization. These elements are embedded into the agent's operational context at runtime. Although the prompt is included in the same file, it remains logically separated to allow easier modification and reuse.

CRANE also implements a human-in-the-loop capability, enabling a human user to replace any AI agent in the pipeline. This is achieved by specifying both the intent to participate and the role to be substituted. Upon activation, CRANE routes the corresponding prompt to the human participant, allowing him to perform the selected role within the review workflow—whether as a Reviewer, Moderator, or Developer.

3.3.5 Challenges, Innovations & Design Properties

One of the primary challenges encountered during CRANE's development was related to dataset constraints. Specifically, the code associated with the Change Requests (CRs) in the dataset could not be executed directly. This limitation made it impossible to use dynamic analysis tools such as SonarCloud or CodeQL, both of which require compilable and runnable code. To address this, Semgrep—a static analysis tool capable of pattern-based vulnerability detection without requiring execution—was adopted as an effective substitute. Further details on dataset limitations and tool selection are provided in Section 4.2.2.

Unlike existing tools that typically assist with either communication or technical aspects of the Code Review process, CRANE is designed to fully support the entire pipeline. It combines AI agents that collaborate iteratively across review, summarization, and code-generation phases. Furthermore, CRANE allows for high configurability and direct user involvement through a human-in-the-loop feature, making it more interactive and flexible than static automation tools.

CRANE is built with reusability, adaptability, and scalability in mind:

- **Adaptability:** Agents can be tailored with different roles, MBTI personality

profiles, domain specializations, and contextual instructions, allowing CRANE to be adapted to a variety of tasks or development environments.

- **Reusability:** The modular architecture ensures that CRANE can be applied across different scenarios by simply adjusting configuration files and prompts.
- **Scalability:** The lenght of the response, number of agents and iterations can be increased or reduced based on task complexity. This allows users to scale CRANE's workflow dynamically to meet specific project needs.

CHAPTER 4

Empirical Evaluation

This section outlines the research questions (RQs) guiding the thesis, details the experimental setup, and describes the evaluation procedure used to validate CRANE. To achieve this, we conducted a comparative study leveraging a real-world dataset of change requests from the Gerrit code review platform, as introduced by Mukadam et al. [24].

4.1 Research Objective and Questions

4.1.1 Research Objective

The goal of this study is to develop a framework able to avoid the creation of suboptimal code by simulating a Code Review Process trough LLMs that collaborate to produce more secure code review outcomes.

4.1.2 Research Questions

Our study is guided by the following research questions:

? RQ_1 On Reducing Vulnerabilities.

How do different system configurations impact the system's ability to reduce code vulnerabilities during automated code review?

By answering this, we aim to understand which architectural or behavioral aspects of CRANE's multi-agent system contribute most effectively to the generation of secure code. This includes evaluating how personality traits, informations about vulnerabilities, number of reviewers and maximum number of iterations influence the frequency and severity of vulnerabilities in the code snippets produced. To address this question, we investigate the following subquestions:

- **RQ1.1**: What is the effect of enabling personality traits on the number of resolved vulnerabilities?
- **RQ1.2**: How does access to additional vulnerability information influence the model's ability to resolve vulnerabilities?
- **RQ1.3**: How does varying the number of agents (2 vs. 3) impact the effectiveness of vulnerability resolution?
- **RQ1.4**: How does increasing the number of review iterations (from 1 to 4) affect the reduction of vulnerabilities?

? RQ_2 On Semantic Alignment.

How do different system configurations affect the semantic alignment of the review outcomes with those of human developers, as measured by cosine similarity?

By answering this, we aim to understand which architectural or behavioral aspects of CRANE's multi-agent system most influence the semantic similarity between its generated code and human-written code. This includes evaluating the role of personality traits, vulnerability information, the number of reviewers, and the maximum number of discussion iterations. To address this question, we investigate the following subquestions:

- **RQ2.1**: To what extent do agent personality traits affect the semantic similarity of the output with human-authored code?

- **RQ2.2**: To what extent does access to additional vulnerability information influence the semantic similarity between CRANE-generated and human-written code snippets?
- **RQ2.3**: How does the number of reviewers (2 vs. 3) impact the semantic alignment of CRANE's output with human-authored code?
- **RQ2.4**: What is the effect of increasing the number of review iterations (1 to 4) on the semantic similarity to human-written code snippets?

> **? RQ_3 On Vulnerability Severity.**
> *How do different system configurations affect the severity of vulnerabilities in the code generated by CRANE?*

By answering this, we aim to understand which architectural or behavioral aspects of CRANE's multi-agent system most influence the severity of remaining or introduced vulnerabilities in the generated code. This includes evaluating the impact of personality traits, additional vulnerability information, the number of reviewers, and the maximum number of discussion iterations. To address this question, we investigate the following subquestions:

- **RQ3.1**: To what extent do agent personality traits influence the severity of the vulnerabilities in the final output?
- **RQ3.2**: Does providing detailed vulnerability information reduce the severity of vulnerabilities in CRANE-generated code?
- **RQ3.3**: How does changing the number of reviewers (2 vs. 3) affect the severity level of remaining or introduced vulnerabilities?
- **RQ3.4**: What is the impact of increasing the number of review iterations (1 to 4) on the severity of vulnerabilities in the final code?

> **? RQ_4 On Human-AI Comparison.**
> *How does the best-performing configuration of CRANE compare to human-written code in terms of vulnerability reduction?*

By answering this, we aim to assess the practical viability of CRANE as a supportive tool in real-world Code Review Processes by directly comparing the vulnerability frequency of CRANE-generated code and human-written code.

4.2 Research Process

This section presents an overview of the research process, as shown in Figure 4.1. Beginning with the selection and sampling of CRs, the inputs are fed into CRANE to generate seven distinct outputs—one for each configuration of the sequential ablation study, as described in Subsection 4.2.5. These outputs are then used to answer the corresponding RQs.

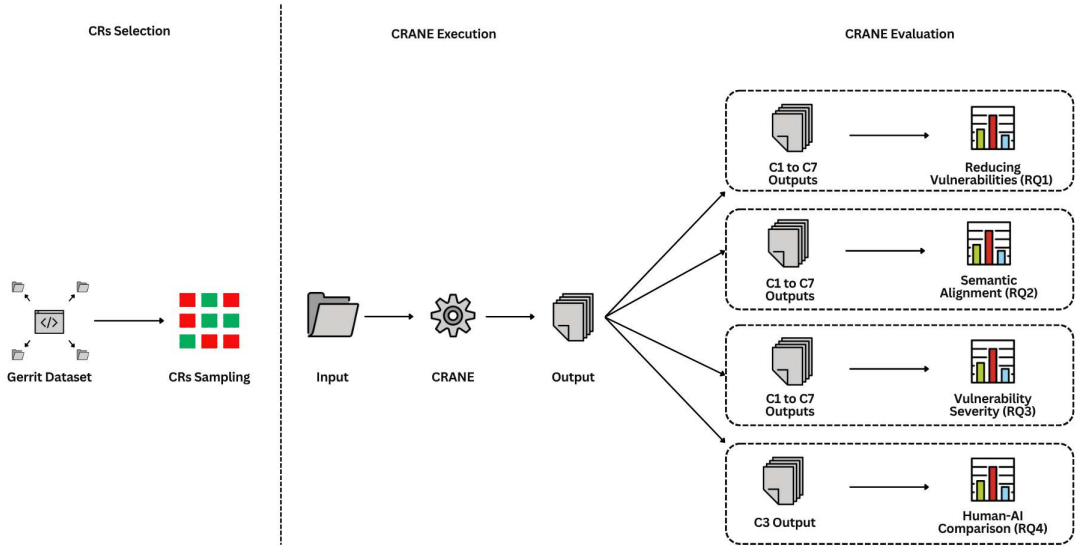

Figure 4.1: CRANE Evaluation Process

4.2.1 Dataset Selection and Preparation

As highlighted by Basic et al. [31], Large Language Models (LLMs) suffer from a phenomenon known as the **self-repair blind spot**: they perform poorly when repairing their own generated code but can fix up to 60% of insecure code produced by other LLMs. To accurately evaluate CRANE's capabilities while mitigating this intrinsic limitation, we selected a dataset able to minimize the presence of LLM-generated code without sacrificing the realism of the evaluation.

For this purpose, we chose the dataset presented by Mukadam et al. [24] because:

- It consists of change requests extracted from **real-world projects**;
- These change requests are sourced from Gerrit, a platform which, due to the period of the extracted CRs, substantially **reduces the likelihood of including AI-generated code**.

From this dataset, a total of 31.610 change requests spanning multiple programming languages were extracted. Following [22, 35] who highlighted the difficulty LLMs face in repairing Java code (with Codex repairing only 20.4% of vulnerabilities), we focused on Java snippets. This filtering yielded **8,167 Java-specific change requests**.

Additionally, because the original change requests varied significantly in size—from 19 to 60,000 tokens—we applied a length filter, limiting the maximum CR size to 2,000 tokens. This resulted in a dataset of 7,410 Java change requests, suitable for evaluating CRANE's performance.

4.2.2 Vulnerability Labeling and Validation

Due to the non-executable nature of the code snippets extracted from Gerrit—being partial segments of classes or methods—traditional dynamic analysis tools could not be used to assess their quality or security. To overcome this limitation, we employed Semgrep, a static analysis tool that detects vulnerabilities based on syntactic and semantic patterns, without requiring code execution.

In addition to Semgrep's default ruleset, we extended its capabilities by incorporating open source detection rules targeting vulnerabilities from both the MITRE

CWE Top 25 Most Dangerous Software Weaknesses and the OWASP Top 10, obtaining a total of **2522 rules**[1][2]. This ensured broad coverage of critical security flaws, improving the robustness of the vulnerability identification process.

4.2.3 Evaluation Metrics

To comprehensively assess CRANE's performance, three core metrics were defined:

- **Vulnerability Count (quantitative metric):** The total number of vulnerabilities identified within each code snippet;
- **Vulnerability Severity (qualitative metric):** The severity level (e.g., low, medium, high) of each detected vulnerability;
- **Semantic Similarity (quantitative metric):** The cosine similarity between the original code snippet and the code generated by CRANE. This metric served to ensure that the code produced by the model didnt diverge significantly from the human-written code.

While both the vulnerability count and severity scores were derived using Semgrep's rule-based detection, cosine similarity was computed using the `sklearn` Python library.

4.2.4 Research Questions Evaluation

> **? RQ_1 — On Reducing Vulnerabilities.**
> *How do different system configurations impact the system's ability to reduce code vulnerabilities during automated code review?*

To answer RQ_1 and its four sub-questions, we evaluated the number of vulnerabilities found in the code produced by each CRANE configuration. The output of each configuration was automatically integrated into a CI/CD pipeline where Semgrep

[1] https://semgrep.dev/p/cwe-top-25
[2] https://semgrep.dev/p/owasp-top-ten

was used to detect vulnerabilities. Since the focus was on comparing configurations, no direct comparison with human-written code was required for this analysis.

> **? RQ$_2$ — On Semantic Alignment.**
> *How do different system configurations affect the semantic alignment of the review outcomes with those of human developers, as measured by cosine similarity?*

For RQ$_2$, we computed the cosine similarity between the CRANE-generated code and the original human-written snippets. This was done using the `sklearn` Python library, which returns a similarity score between 0 (completely dissimilar) and 1 (identical). The mean cosine similarity for each configuration was calculated using Equation 4.2.1:

$$\mu_{\text{cosine}} = \frac{1}{n} \sum_{i=1}^{n} \text{cosine}(CR_i^{\text{human}}, CR_i^{\text{CRANE}}) \tag{4.2.1}$$

Where:

- CR_i^{human} denotes the human-written code snippets for the i^{th} change request;
- CR_i^{CRANE} denotes the corresponding code snippets generated by CRANE;
- n represents the total number of change requests.

> **? RQ$_3$ — On Vulnerability Severity.**
> *How do different system configurations affect the severity of vulnerabilities in the code generated by CRANE?*

To answer RQ$_3$ and its four sub-questions, we analyzed the severity levels of vulnerabilities detected in the output of each CRANE configuration. Each configuration's output was processed through an automated CI/CD pipeline using Semgrep, which not only flagged the presence of vulnerabilities but also categorized them by severity (e.g., low, medium, high). This analysis allowed us to assess whether certain configurations were more prone to introducing or retaining high-severity vulnerabilities, even if the total number of vulnerabilities was reduced.

> **? RQ_4 — On Human-AI Comparison.**
> *How does the best-performing configuration of CRANE compare to human-written code in terms of vulnerability reduction?*

After completing the sequential ablation study, we identified the best-performing CRANE configuration based on the reduction of vulnerabilities. We then directly compared the outputs from this configuration against the original human-written code to evaluate CRANE's effectiveness in producing more secure code. The effectiveness of each configuration in reducing vulnerabilities is computed using Equation 4.2.2.

$$\text{Effectiveness (\%)} = 100 - \left(\frac{\text{Vulnerabilities in Configuration } n \times 100}{\text{Vulnerabilities in Final Dataset}} \right) \tag{4.2.2}$$

Where:

- **Vulnerabilities in Configuration *n*** is measured by counting the vulnerabilities in the change requests produced at the end of the CRANE pipeline.
- **Vulnerabilities in Final Dataset** is measured by counting the vulnerabilities in the human-resolved change requests provided by the Gerrit Dataset [24].

4.2.5 CRANE's Configurations

To systematically evaluate CRANE's performance and adaptability, a number of configuration parameters were varied. These parameters were selected based on both theoretical relevance and empirical support from prior research. The following dimensions were explored:

- **Personality Type:** Each Reviewer was assigned a specific MBTI personality type. These types were selected based on findings from multiple studies that have linked personality traits to performance in software engineering tasks [23, 44, 43]. This variable was binary (enabled/disabled).
- **Vulnerability Information:** In some configurations, Reviewers were given additional context about known or suspected vulnerabilities in the code snippet.

This included both the name of the vulnerability (e.g., "SQL Injection") and a short textual description. This variable was binary (enabled/disabled).

- **Number of Reviewers:** We experimented with different team sizes, specifically using 2 and 3 Reviewers. A single-agent configuration was excluded, as code reviews inherently benefit from collaboration.
- **Number of Iterations:** CRANE was evaluated using a variable number of review iterations, ranging from 1 to 4, to assess the impact of iterative refinement on the quality of generated code. Importantly, **completing all iterations was not mandatory**. If all the Reviewers reached a consensus indicating that further discussion was unnecessary or redundant, the review process could terminate early, allowing the system to simulate realistic reviewer convergence and avoid unnecessary computation.

Other parameters, such as the number of messages per iteration and the maximum token limit per agent, were kept constant across all experimental runs. The number of messages was fixed at 3, which represents the minimum required to form a coherent conversational exchange. This includes an initial message that typically does not account for prior input from other agents, followed by two additional messages that allow for **context-aware interaction**. Moreover, parameters such as the model temperature were not modified, as the objective was to evaluate the default behavior of the models in their base configuration. This choice ensures that the results reflect the inherent capabilities of the models, without the influence of additional tuning or stochastic variation.

Prompts used for each agent were defined and refined during the iterative design phase following the Design Science Research Methodology (DSRM). To clarify the configuration space, Table 4.1 summarizes the variables and their tested values.

To evaluate CRANE across different configurations while limiting the total number of experimental runs, we adopted a **sequential ablation strategy** on CRANE's fully automated pipeline, shown in Table 4.2. This approach incrementally introduced one configuration variable at a time—following the order presented in Table 4.1—to isolate and better understand the individual impact of each variable on performance.

Table 4.1: Overview of CRANE Configuration Parameters

Type	Characteristics
Personality Type	Enabled / Disabled
Vulnerability Info	Enabled / Disabled
Number of Reviewers	2, 3
Number of Iterations	1, 2, 3, 4
Messages per Iteration	Fixed (3 per Reviewer)
Token Limit per Reviewer	Fixed (500 tokens)
Token Limit per Moderator	Fixed (1000 tokens)
Token Limit per Developer	Fixed (3000 tokens)
Prompt Design	Refined during DSRM iterations

Table 4.2: Sequential Ablation Study

Configuration	Personality	Security Info	Reviewers	Iterations
C1	Disabled	Disabled	2	1
C2	Enabled	Disabled	2	1
C3	Best one	Enabled	2	1
C4	Best one	Best one	3	1
C5	Best one	Best one	Best one	2
C6	Best one	Best one	Best one	3
C7	Best one	Best one	Best one	4

This method allowed us to specifically evaluate the impact of personality traits and vulnerability information on the effectiveness of change request (CR) resolution. For each configuration, performance was primarily assessed based on the **number of reduced vulnerabilities**. Severity was only considered when two configurations (one of which had to be the current best-performing) produced the same number of vulnerabilities; otherwise, the total count served as the main comparison metric.

Additionally, cosine similarity between the original and generated snippets was

computed as a supplementary indicator to evaluate semantic consistency, rather than as a primary measure of effectiveness.

4.2.6 Computational Environment

To ensure reproducibility and provide transparency regarding the technical setup, this section details the hardware and software configuration used during the development, execution, and evaluation of CRANE. Table 4.3 summarizes the key components of the computational environment, including processing capabilities, operating system, programming language version, and tools employed for large language model access and static analysis.

Table 4.3: Hardware and Software Environment for CRANE

Component	Specification
Processor	Intel® Core™ Ultra 7 155H, 3.80 GHz, 16 cores, 22 logical processors
GPU	NVIDIA GeForce RTX 4060 Laptop GPU
Operating System	Windows 11
Python Version	3.12.9
Key Libraries	All required Python packages and dependencies are listed in the project's README file [45]
Model Access	CRANE interacts with LLMs via the OpenAI API (gpt-4o-mini-2024-07-18), orchestrated through custom Python scripts
Static Analysis Tool	Semgrep was used for vulnerability detection and severity classification

4.2.7 Reproducibility and Open Resources

To ensure the reproducibility of our experiments and promote transparency, the following resources will be released:

- The **dataset splits** used for each configuration
- The **exact values of each configuration parameter**
- The **prompt templates** assigned to each agent role

All materials will be made publicly available trough Zenodo platform [25, 45].

4.2.8 Ensuring Fairness and Reliability

To ensure a fair and reliable evaluation, all system configurations were tested under identical runtime conditions, varying one parameter at a time through a **controlled sequential ablation study**. The dataset—composed of real-world, developer-written change requests extracted from the Gerrit code review platform—was selected to minimize the presence of LLM-generated code and to reflect realistic development environments. Vulnerability detection was performed using Semgrep with standardized rule sets for the MITRE CWE Top 25 and OWASP Top 10. Cosine similarity was computed using the `sklearn` library. Each configuration was evaluated across 366 filtered Java CRs, a statistically significative dataset of the original 7410 CRs, and performance metrics were averaged to reduce outlier influence and improve statistical reliability.

CHAPTER 5

Results

5.1 Overview of Key Findings

Before presenting the results in detail, we summarize the most relevant findings from CRANE's fully automated pipeline, as no humans were included during the various executions. Overall, while the human code review process resulted in an **86.84% increase in vulnerabilities**, CRANE achieved a reduction of medium-severity vulnerabilities by **85.91%** across generated code snippets.

Interestingly, the inclusion of additional information—such as reviewer personality types and explicit vulnerability descriptions—enhanced CRANE's ability to generate more secure code. However, this improvement came at the cost of reduced semantic similarity with human-written code.

Moreover, increasing the number of reviewers or iterations led to a decline in both security performance and semantic alignment. This highlights the importance of careful configuration when optimizing CRANE for specific outcomes. Overall results, including the Starting Dataset (i.e., the model's input) and the Final Dataset (i.e., the human-resolved CRs), are shown in Table 5.1. At last, vulnerability severity did not affect the selection of the best-performing configuration, as there were no cases where it tied with another configuration in terms of vulnerability count.

Table 5.1: Overall results accross all the configurations

Type	Costs	Number of vulnerabilities	Cosine similarity (mean)
Starting Dataset	–	38	–
Final Dataset	–	71	–
Configuration 1	1.03 €	56	0.706
Configuration 2	1.36 €	52	0.701
Configuration 3	1.06 €	10	0.691
Configuration 4	1.54 €	11	0.686
Configuration 5	2.28 €	15	0.655
Configuration 6	2.67 €	15	0.617
Configuration 7	5.10 €	21	0.602

5.2 RQ1: Results on Code Vulnerability Reduction

As presented in Table 5.1 and illustrated in Figure 5.1, the number of vulnerabilities identified across the various CRANE configurations varied considerably. Notably, all configurations achieved better results than the human-written baseline, confirming that even basic CRANE setups can enhance security outcomes. Most of the vulnerabilities detected in all variants corresponded to known categories in the OWASP Top 10 and the MITRE CWE Top 25, underscoring the practical relevance of the issues addressed.

To understand how specific design changes influenced these results, it is useful to examine the evolution of CRANE's configurations in relation to the four sub-research questions.

The **first enhancement** involved the introduction of personality traits to the reviewing agents. This addition aimed to simulate more human-like reviewing dynamics, and it resulted in a modest reduction in vulnerabilities—from 56 to 52. However, personality alone was not sufficient to significantly enhance performance.

A much more substantial shift occurred with the **second configuration**, in which agents were also provided with explicit vulnerability-related information. This

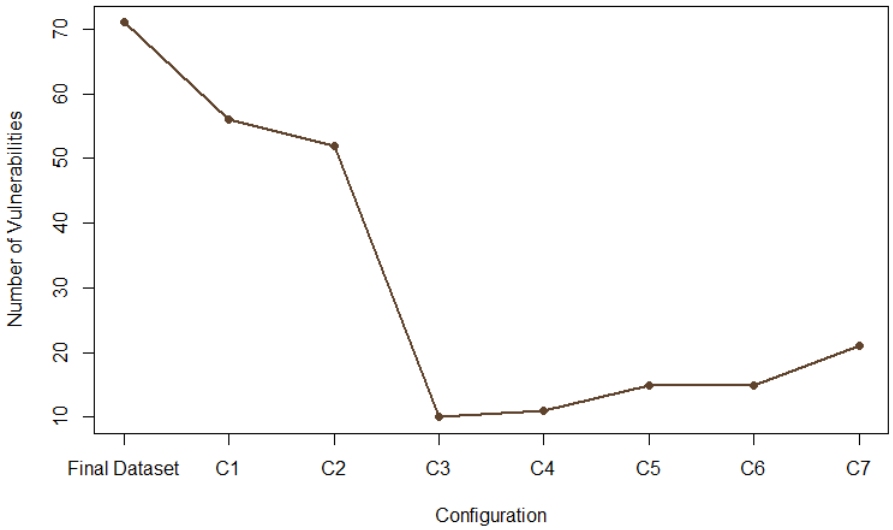

Figure 5.1: Number of Vulnerabilities across Configurations

change led to a sharp drop in vulnerabilities, from 52 to just 10. Orienting the agents toward particular classes of vulnerabilities, the system became markedly more effective at resolving security issues.

The **third modification** involved increasing the number of agents participating in each review from two to three. Contrary to expectations, this adjustment resulted in a slight increase in the number of vulnerabilities (from 10 to 11). Therefore, simply scaling the number of reviewers does not necessarily translate into improved outcomes, and that collaborative mechanisms must be carefully calibrated.

Finally, the **fourth configuration** allowed for extended review iterations—up to four rounds instead of a single pass. Surprisingly, this led to a degradation in performance, with the total number of vulnerabilities increasing to 21.

RQ_1 On Reducing Vulnerabilities.

System configurations that include personality traits and explicit vulnerability information significantly enhance CRANE's ability to reduce code vulnerabilities. In contrast, configurations involving additional reviewers or unnecessary iterations tend to degrade performance by reintroducing or failing to resolve certain vulnerabilities.

5.3 RQ2: Results on Semantic Similarity

As shown in Table 5.1 and Figure 5.2, cosine similarity between the CRANE-generated code and the original human-written snippets declined progressively across configurations. Even in the baseline setup (C1), where no enhancements were added beyond the initial prompt and contextual input, the average similarity was already somewhat reduced.

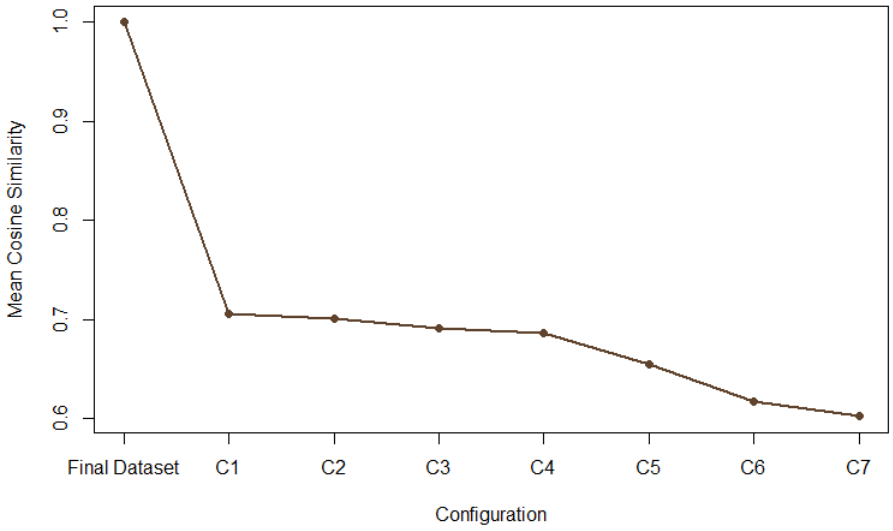

Figure 5.2: Mean Cosine Similarity across Configurations

The configuration that achieved the highest semantic alignment was indeed C1, with a cosine similarity score of 0.706. This configuration essentially reflects the most direct and minimally manipulated interaction with the LLMs

In contrast, the **introduction of personality traits** in C2 resulted in a slight drop in similarity to 0.701. Although personality-based agents offer more realistic and diverse reviewing styles, this shift indicates that such personalization may not necessarily produce outputs that resemble human-written code in structure or syntax.

The **inclusion of explicit vulnerability information** in C3 reduced the score further to 0.691. This decrease reflects the transformative effect of task-specific guidance: when agents are pushed to optimize for security, the resulting code may tend to diverge from the original, as it involves structural modifications, reorganization, or the insertion of defensive programming constructs not present in the baseline.

This trend continued as configurations grew more complex. In C4, which added a **third reviewing agent**, similarity declined to 0.686.

Finally, C5—C7, featuring additional, **non-mandatory review iterations**, exhibited the most significant decrease, with C7 having a similarity of just 0.601. This represents a 0.105 drop compared to the highest value observed in C1. The reduced similarity in this configuration likely reflects the cumulative effect of multiple revisions.

RQ_2 On Semantic Alignment.

Semantic alignment—measured via cosine similarity—was highest when only the base prompt and context were used. The introduction of additional elements such as personality traits, vulnerability descriptions, or increased review complexity (e.g., additional agents or iterations) led to a progressive decrease in similarity.

5.4 RQ3: On Vulnerability Severity

While C3 stood out in terms of overall vulnerability reduction, a deeper look at vulnerability severity reveals a more nuanced picture. Notably, none of the configurations enabled CRANE to resolve the **high-severity vulnerabilities** that were originally present in the dataset. To understand how each configuration influenced the severity of vulnerabilities, we examined the evolution of severity levels across the seven configurations compared to the Final Dataset, as shown in Table 5.2. In particular, we observed how changes in design parameters affected the prevalence of both high and medium-severity vulnerabilities.

C1 already introduced a slight decrease in medium-severity issues (from 69 to 53) and maintained the original two high-severity cases. When **personality traits were enabled** in C2, the number of medium-severity vulnerabilities was reduced modestly, dropping to 50.

The most impactful configuration in this regard was C3, where **vulnerability-specific information was provided** to the agents. This enhancement drastically reduced medium-severity vulnerabilities—from 50 to just 8—without introducing any additional high-severity ones.

However, as **more agents** were introduced in C4, we began to observe signs of diminishing returns. Although no high-severity vulnerabilities emerged, the number of medium-severity vulnerabilities crept up slightly from 8 to 9.

The most concerning trend was observed in C5 to C7, which introduced additional, **non-mandatory review iterations**. While the intention was to simulate extended deliberation, the outcome was counterproductive: 9 new high-severity vulnerabilities were introduced—raising, in C7, the total from 2 to 11—without further gains in reducing medium-severity issues.

Interestingly, across all variants and the original Gerrit dataset, there were no vulnerabilities categorized as either critical or low severity, narrowing the focus of this evaluation to the mid-to-high severity spectrum.

Table 5.2: Vulnerabilities' Severity Among Different Datasets

Type	Critical	High	Medium	Low
Final Dataset	0	2	69	0
Configuration 1	0	3	53	0
Configuration 2	0	2	50	0
Configuration 3	0	2	8	0
Configuration 4	0	2	9	0
Configuration 5	0	6	9	0
Configuration 6	0	3	12	0
Configuration 7	0	11	10	0

RQ_3 On Vulnerability Severity

While the inclusion of personality modeling and vulnerability information reduced medium-severity vulnerabilities without introducing new high-severity ones, extended review iterations led to the emergence of up to nine additional high-severity vulnerabilities.

5.5 RQ4: Human-AI Comparison

5.5.1 Statistical Analysis of Configurations

To determine the best-performing configuration from a statistical perspective, we conducted a **comparative analysis of each configuration's output**, including the Gerrit Dataset of change requests (CRs). Each dataset was transformed into a list where each item represented the number of vulnerabilities found in a given CR—encoded as either 0 (no vulnerabilities) or x (number of vulnerabilities detected).

We first applied the **Friedman test** to assess whether statistically significant differences existed among the datasets. The resulting p-value (< 0.000) led to the rejection of the null hypothesis (H_0), confirming that at least one dataset differed significantly. As a follow-up, a post-hoc **Paired Permutation Test** was performed, and the statistically significant results are summarized in Table 5.3.

Notably, no statistically significant difference was found between configurations C3 and C4 (p-value = 1), or between C3 and configurations C5, C6, and C7. This suggests that these configurations may be interchangeable in terms of performance. However, C3 outperformed all others in terms of both total vulnerability reduction and severity classification, and was therefore selected as the best-performing configuration.

In contrast, both C3 and C4 differed significantly from configurations C1 and C2. They also exhibited statistically significant differences—albeit with higher p-values—when compared to the Gerrit dataset, which was divided into two subsets: the **Starting Dataset** and the **Final Dataset**.

5.5.2 Best-Performing Configuration: C3

Based on the previous analysis, C3 was identified as the best-performing setup. It integrates personality modeling and explicit vulnerability information.

C3 achieved a **85.91% reduction in vulnerabilities** compared to the original code and demonstrated a statistically significant improvement over the Final Dataset (p-value = 0.0015). Importantly, this performance did not come at the cost of semantic coherence: the configuration maintained a high **cosine similarity score of 0.691**.

Table 5.3: Paired Permutation Test Results

Group 1	Group 2	p-value
C4	C1	0.0000
C3	C2	0.0000
C4	C2	0.0000
C6	C2	0.0001
C3	C1	0.0002
C6	C1	0.0002
C5	C2	0.0008
C5	C1	0.0011
C3	Final Dataset	0.0015
C4	Final Dataset	0.0019
C3	Starting Dataset	0.0051
C4	Starting Dataset	0.0053
C5	Final Dataset	0.0069
C6	Final Dataset	0.0071
C7	C2	0.0158
C7	C1	0.0160
C7	Final Dataset	0.0381
C6	Starting Dataset	0.0436
C5	Starting Dataset	0.0472

These results indicate that CRANE's fully automated, multi-agent pipeline can generate substantially more secure code while preserving alignment with human intent and structure. A visual representation of these findings is provided in Figure 5.3.

RQ_4 On Human-AI Comparison.

CRANE's best-performing configuration (C3), which combined LLMs enriched

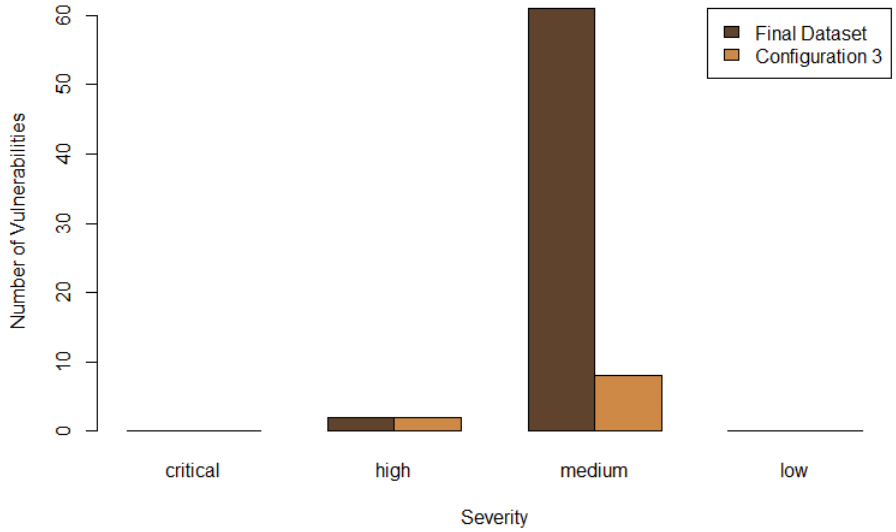

Figure 5.3: Human-AI Vulnerabilities Comparison

> with personality traits and vulnerability information, achieved an 85.91% reduction in code vulnerabilities compared to the human-written baseline, while maintaining a cosine similarity of 0.691.

5.6 A Practical Example

To illustrate the practical effectiveness of CRANE, we analyze an example from code review `CR_15985-2`. This review includes a code snippet containing a medium-severity vulnerability classified as `active-debug-code-printstacktrace`. According to the vulnerability definition: "*Possible active debug code detected. Deploying an application with debug code can create unintended entry points or expose sensitive information.*". Code snippets are shown in Table 5.4.

In the original (human-written) code, the line using `printStackTrace()` directly exposes internal exception details, which is considered a poor security practice in production environments. In contrast, CRANE's generated version replaces this insecure pattern with structured logging via the Android `Log` class. This approach preserves the necessary debugging information while avoiding the exposure of sensitive system internals. By eliminating active debug code, CRANE adheres to secure coding standards and effectively mitigates the identified vulnerability.

Table 5.4: Example of Vulnerability Resolution by CRANE

File	Code Snippet

Human-Written Code	`catch(IOException e) {`
	`  e.printStackTrace();`
	`}`

CRANE-Generated Version	`catch (IOException ioe) {`
	`  Log.e("MonkeyRunner", "Unable to get frame buffer: " + ioe.getMessage());`
	`}`

CHAPTER 6

Discussion

6.1 Overview of Findings

Although results varied across system configurations, the best-performing setup (C3) demonstrated CRANE's potential in **improving software security**. It reduced the number of detected vulnerabilities by **85.91%**, introduced no high-severity vulnerabilities, and achieved a cosine similarity of 0.691 compared to the original human-written code. This suggests that CRANE not only enhances code security but also preserves a strong degree of semantic alignment with developer code.

Regarding RQ1, RQ2 and RQ3, results reveal that incorporating personality traits and vulnerability-related information improves vulnerability reduction, but adding more reviewers or multiple interaction rounds tends to have a detrimental effect on both vulnerability reduction, semantic alignment and vulnerability severity. In particular, C7—the worst-performing in terms of vulnerability severity—managed to resolve only **70.42%** of vulnerabilities and achieved a cosine similarity of 0.602. Notably, 11 of the 21 vulnerabilities were of high severity, indicating that overcomplicating the review process may introduce serious security vulnerabilities.

Although the existing literature offers limited in-depth comparisons between large language models (LLMs) and human developers in generating secure code [32, 46],

such analyses are even scarcer when focusing specifically on Java. Nevertheless, Wu et al. [35] suggest that most LLMs and Automated Program Repair (APR) tools—except Codex—are primarily effective in resolving simple vulnerabilities in Java (e.g., deleting lines or renaming variables) and often struggle with more complex Common Weakness Enumeration (CWE) patterns, such as CWE-172 (Encoding Error), CWE-325 (Missing Cryptographic Step), and CWE-444 (HTTP Request Smuggling).

CRANE, however, appears capable of addressing security issues. That said, since the initial dataset did not include these specific CWEs, a comprehensive comparison with existing approaches remains out of scope.

6.2 Discussion per Research Question

In this section, the results for each Research Question (RQ) will be analyzed in greater detail, with the goal of uncovering insights that may inform future research directions. At the same time, to extract further insights, an additional analysis has been conducted—specifically, a comparison between the last and penultimate iterations of CRANE's outputs. Since the number of iterations is not fixed and may vary depending on the reviewers' decisions, some CRs may contain only a single response. In such cases, that single response will be considered as both the final and penultimate iteration. This approach ensures comprehensive coverage of all CRs and includes every vulnerability present in the corresponding code snippets.

6.2.1 RQ1: On Code Vulnerability Reduction

Although each configuration led to a reduction in vulnerabilities compared to the original code, C3 emerged as the most effective. This configuration combined personality traits with detailed vulnerability information—extracted via Semgrep—including both the vulnerability name and a descriptive explanation. These additional layers of context appear to play a key role in enhancing CRANE's ability to generate more secure code.

This finding—especially the addition of vulnerability information—aligns with observations by Yetiştiren et al. [47], who showed that, for Python, detailed doc-

umentation (e.g., docstrings) in the input significantly enhances both the validity and correctness of AI-generated code. In their study, **validity** referred to syntactic correctness, while **correctness** was measured by the proportion of test cases passed.

For Java code specifically, Nguyen et al. [48] reported that Copilot tends to perform better in well-defined programming scenarios, especially when prompts are appropriately crafted. Their findings also indicate that suggestions in Java are generally of low complexity and comparable in understandability to those in Python and JavaScript, possibily suggesting that CRANE could be therefore used also on different programming languages. Moreover, these insights further support the notion that well-structured, informative input—such as CRANE's enriched prompts—can lead to more secure and reliable AI-generated code.

Other than Dos Santos et al. [41] who state that a number of two active reviewers seems to be the otptimal case considering a trade-off between duration and contribution from reviewers, the reason behind the performance degradation caused by the addition of reviewers remains unclear. Meanwhile, the negative impact of additional iterations—specifically, the observed decline in vulnerability reduction and the increased introduction of high-severity vulnerabilities—may be linked to an intrinsic limitation known as the **self-repair blind spot**. This phenomenon suggests that LLMs tend to perform worse when attempting to fix code they have generated themselves, compared to repairing code authored by other LLMs. To explore this further, we conducted a focused analysis on C6 and C7, which involved the highest number of iterations. The following observations were made:

- **Configuration 6:** The penultimate iteration showed 14 vulnerabilities, while the final iteration had 15. Notably, the penultimate version contained only one high-severity vulnerability, whereas the final output introduced three. Two of the vulnerabilities that reappeared in the final version had already been resolved earlier and were originally present in the Starting Dataset. This suggests that the last iteration not only failed to improve but actually reintroduced previously resolved vulnerabilities—strongly supporting the self-repair blind spot hypothesis and underscoring the potential need for human oversight in iterative automated code reviews. Results are shown in Table 6.1

- **Configuration 7:** The penultimate iteration contained 24 vulnerabilities, whereas the final output contained 21. In both iterations, CRANE introduced 11 high-severity vulnerabilities. This suggests that the final iteration resolved some issues but failed to reduce the number of hihg-severity issues, indicating a limited benefit of additional iterations. Results are shown in Table 6.2,

Table 6.1: Penultimate and Last Iteration for C6

Severity	Penultimate	Last
Critical	0	0
High	1	13
Medium	3	12
Low	0	0

Table 6.2: Penultimate and Last Iteration for C7

Severity	Penultimate	Last
Critical	0	0
High	11	11
Medium	13	10
Low	0	0

Results from RQ1—particularly the addition of contextual information—support prior studies [47, 48, 41], which emphasize that richer prompts help LLMs produce better code. This suggests that an initial round of automated suggestions may enhance the model's ability to handle tasks that would otherwise require human input, enabling a fully autonomous workflow.

On Code Vulnerabilities

Adding extra iterations or reviewers without a clear rationale can hinder rather than enhance performance. To optimize CRANE's effectiveness in reducing vulnerabilities, its configuration—particularly the number of reviewers and review rounds—should be tailored to the complexity and severity of each Change Request (CR). This suggests that minimizing unnecessary steps in the automated pipeline helps preserve both efficiency and output quality.

6.2.2 RQ2: On Semantic Alignment with Human Code

In contrast to RQ1, the configuration that achieved the highest semantic alignment with the original human-written code was C1, which included no additional information about vulnerabilities or personality traits, indicating that the inclusion of further contextual elements tended to reduce the cosine similarity between the CRANE-generated code and the reference dataset. Importantly, no trade-off between security and semantic similarity was considered, as RQ2 focused solely on ensuring that the generated output did not deviate drastically from the original intent and structure of the human-written code. More on that will be presented in Section 6.3

To further examine the effects of iterative review on semantic alignment, C6 and C7—the two setups with the highest number of iterations—were analyzed in greater detail. Unlike in RQ1, where these configurations had varying impacts on vulnerability reduction, both showed a decline in cosine similarity from the penultimate to the final iteration. Specifically, in C6, similarity dropped from 0.644 to 0.617, and in C7, from 0.614 to 0.602. This trend suggests that the more a code snippet is modified by LLMs, the less it resembles human-written code. Although these models are trained on human code, this result may point to a limitation in their ability to preserve human-like coding style over multiple iterations.

On Semantic Alignment

Providing additional contextual information—such as personality traits or detailed vulnerability descriptions—to reviewers may unintentionally reduce the semantic alignment between generated and human-written code. This suggests that, if the goal is to produce code that closely mimics human style and structure, limiting the number of iterations and keeping reviewer prompts simple may be more effective.

6.2.3 RQ3: On Vulnerability Severity

While clear patterns emerged in terms of vulnerability count and cosine similarity—indicating direct or indirect correlations—the analysis of vulnerability severity painted a more nuanced picture. Personality traits, enriched vulnerability information, and the presence of additional reviewers didn't lead to an increase in high-severity vulnerabilities. However, configurations involving multiple iterations correlated with a noticeable rise in such vulnerabilities; notably, C7 introduced 11 high-severity issues. This trend suggests that LLMs often fail to prioritize security, a behavior also observed by Wang et al. [39], who noted that current models frequently overlook critical security risks during code generation, leading to vulnerable outputs.

Previous studies also indicate that LLMs tend to introduce certain types of CWEs more frequently than others. For instance, Fu et al.[46] report that CWE-330 (Use of Insufficiently Random Values), CWE-94 (Improper Control of Code Generation), and CWE-79 (Cross-site Scripting) commonly appear in Copilot-generated code snippets. Similarly, a systematic literature review by Basic et al.[31] found that injection vulnerabilities are among the most frequently introduced by LLMs, with 16 out of 20 analyzed studies reporting this pattern.

To assess how CRANE's vulnerability profile compares with existing results, Figure 6.1 contrasts the **Final Dataset**—i.e., the human-resolved change requests from the Gerrit dataset—with CRANE's best-performing configuration (C3). Meanwhile, Figure 6.2 contrasts the two worst-performing configurations in terms of number of vulnerabilities (C1) and their severity (C7). Since multiple vulnerabilities can correspond to the same CWE, Table 6.3 provides a mapping of Semgrep-identified

vulnerabilities to their respective CWE categories for Figure 6.1, while Table 6.4 offers a similar mapping for Figure 6.2.

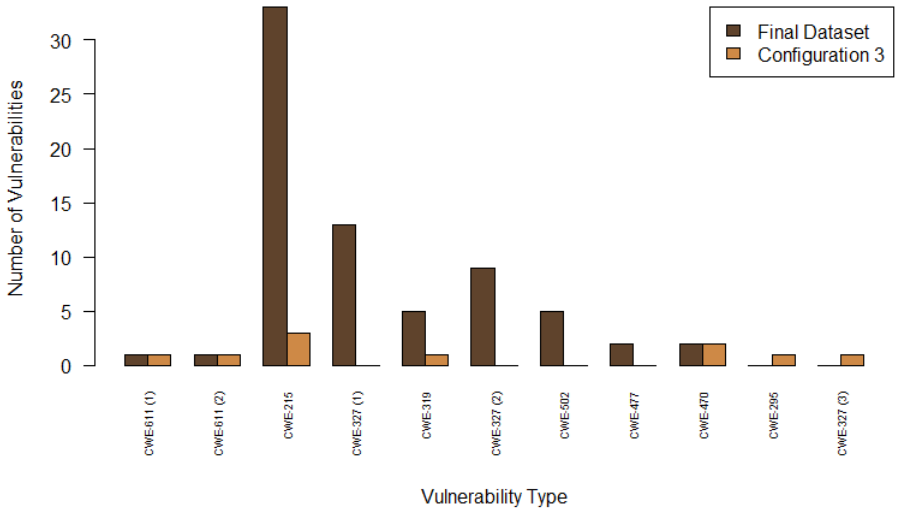

Figure 6.1: Vulnerability Severity: Final Dataset vs Configuration 3

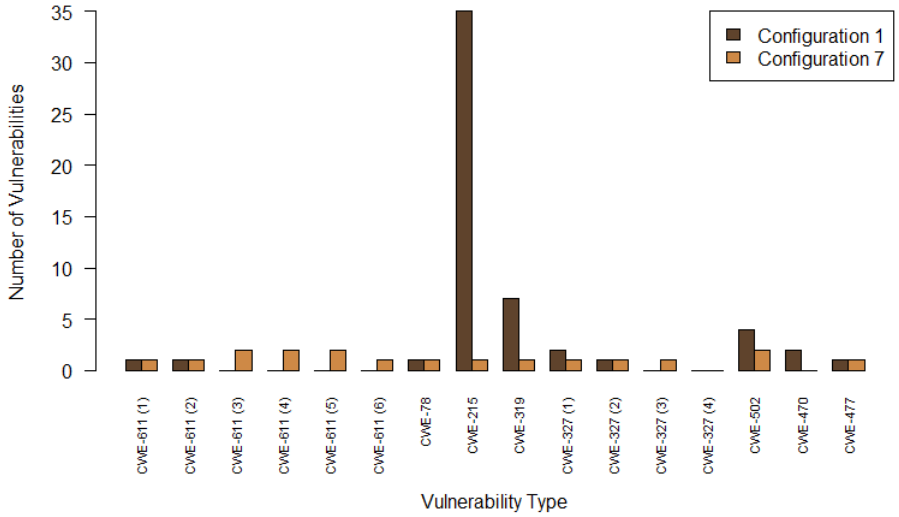

Figure 6.2: Vulnerability Severity: Configuration 1 vs Configuration 7

Based on the data, it becomes evident that the types of vulnerabilities introduced vary depending on the system configuration, suggesting that configuration choices influence not only the quantity but also the nature of security flaws. For instance, in

Table 6.3: Mapped Vulnerabilities to CWE IDs

Semgrep Rule Name	CWE	CWE Label
schemafactory-xxe-schema	CWE-611	CWE-611 (1)
schemafactory-xxe	CWE-611	CWE-611 (2)
active-debug-code-printstacktrace	CWE-215	CWE-215
ecb-cipher	CWE-327	CWE-327 (1)
unencrypted-socket	CWE-319	CWE-319
use-of-aes-ecb	CWE-327	CWE-327 (2)
object-deserialization	CWE-502	CWE-502
defaulthttpclient-is-deprecated	CWE-477	CWE-477
unsafe-reflection	CWE-470	CWE-470
insecure-hostname-verifier	CWE-295	CWE-295
disallow-old-tls-versions1	CWE-327	CWE-327 (3)

both the human-written snippets and **C1**, the most frequently observed vulnerabilities align with CWE-209 and CWE-327.

In contrast, **C3** introduces fewer vulnerabilities overall, but includes several low-frequency, medium-impact types—such as CWE-295 and CWE-319.

C7, on the other hand, introduces multiple vulnerabilities related to XML processing, including CWE-611. This configuration also shows an elevated frequency of CWE-295 and CWE-319 vulnerabilities.

Most CWEs observed align with those reported in the literature—such as CWE-209 and CWE-327, frequently linked to LLMs [46, 31]—except for CWE-470, which, though present in the dataset, is underreported in prior studies. This may indicate dataset-specific trends or a broader underrepresentation. Notably, no critical-severity vulnerabilities were found, suggesting LLMs may have some awareness of critical vulnerabilities to avoid.

Table 6.4: Mapping of Semgrep Rules to CWEs

Rule Name	CWE	CWE Label
schemafactory-xxe-schema	CWE-611	CWE-611 (1)
schemafactory-xxe	CWE-611	CWE-611 (2)
documentbuilderfactory-xxe	CWE-611	CWE-611 (3)
documentbuilderfactory-xxe-parameter-entity	CWE-611	CWE-611 (4)
documentbuilderfactory-disallow-doctype-decl-missing	CWE-611	CWE-611 (5)
transformerfactory-dtds-not-disabled	CWE-611	CWE-611 (6)
command-injection-process-builder	CWE-78	CWE-78
active-debug-code-printstacktrace	CWE-215	CWE-215
unencrypted-socket	CWE-319	CWE-319
ecb-cipher	CWE-327	CWE-327 (1)
use-of-default-aes	CWE-327	CWE-327 (2)
weak-ssl-context	CWE-327	CWE-327 (3)
disallow-old-tls-versions1	CWE-327	CWE-327 (4)
object-deserialization	CWE-502	CWE-502
unsafe-reflection	CWE-470	CWE-470
defaulthttpclient-is-deprecated	CWE-477	CWE-477
httpurlconnection-http-request	CWE-200	CWE-200

On Severity Vulnerability

System configuration not only influences the severity of vulnerabilities introduced but also affects their types. While the majority of CWEs observed align with prior literature on LLM-induced vulnerabilities, the appearance of CWE-470—absent in most other studies—suggests that while multi-agent LLM systems do not dramatically shift vulnerability categories, they may expose less frequently discussed risks depending on context or prompt structure.

6.2.4 RQ4: Human-AI Comparison

When comparing the human-written code to the output generated by CRANE, all configurations demonstrated **improved performance**, with C3 yielding the best results. This configuration—enriched with personality traits and detailed vulnerability descriptions—led to a substantial enhancement in security, achieving an **85.91%** reduction in vulnerabilities. However, it did not resolve the two high-severity issues present in the dataset. In terms of semantic alignment, C3 maintained a strong similarity score of 0.691 with the human-written code, indicating that it effectively preserved the original intent and structure. These findings suggest that CRANE, particularly in its best-performing configuration, is capable of mimicking human coding behavior and reducing vulnerabilities while effectively collaborating with other tools in a multi-agent setup.

Although CRANE has shown to be effective—producing code that is both semantically aligned and significantly more secure—there are important considerations and limitations to acknowledge. Firstly, the dataset used in this study consisted of small, incomplete code snippets (e.g., partial classes or methods), which may have impacted the model's performance. As noted by Yetiştiren et al. [47], LLMs tend to perform better when provided with **well-defined programming scenarios**, which was not always the case here. For instance, if a referenced function was missing from the snippet, the model might struggle to understand its purpose or usage.

This is further supported by Pan et al. [30], who note that both general-purpose and code-specialized LLMs underperform when code context is partial or insufficient—especially in code completion tasks. Without enough context, the model may fail to generate relevant or syntactically correct content.

Husein et al. [49] reinforce this by identifying context comprehension as one of the primary challenges for LLMs. Capturing both local and global code dependencies is critical for accurate code generation. This limitation may vary by programming language. For example, LLMs generally perform better with statically typed languages like Java (used in this dataset) where type information is explicit, while they struggle more with dynamically typed languages like Python.

Since CRANE was still able to perform effectively on incomplete or partial snip-

pets, the results suggest that the initial round of reviewers may have compensated for the missing context—filling in critical gaps that would otherwise hinder the Developer, which is responsible for generating the final code. This may suggest that providing additional insights or guidance early in the process can enable comparable outcomes to those achieved with fully complete inputs. Further investigation is warranted to validate this hypothesis and explore the extent to which LLMs contextual enrichment can substitute for complete code snippets.

A notable issue emerged in configurations with multiple review iterations: LLMs often failed to recognize when a snippet was already complete or needed no further edits. This suggests a limitation in the model's ability to detect completion boundaries. Pan et al. [30] also reported this behavior, noting that many models cannot reliably decide whether to continue or stop the code completion process.

To gain a deeper understanding of how vulnerabilities are handled in the generated code, Table 6.5 presents a breakdown of vulnerabilities that were removed, introduced, or left unchanged by the best-performing configuration (C3). Out of the 38 original vulnerabilities present in the dataset, C3 successfully resolved 34. However, it also introduced 6 new vulnerabilities, while the remaining 4 correspond to issues that were not addressed. These findings suggest that, although CRANE demonstrates strong capabilities in mitigating existing security issues, it may also inadvertently introduce new vulnerabilities during the code generation process. This underscores the need for continued human oversight or further refinement of the system's security reasoning mechanisms. A deeper discussion about AI secure code generation is presented in Subsection 6.2.3.

On Human-AI Comparison

CRANE significantly outperforms human-written code in terms of vulnerability reduction, by removing 34 out of 38 vulnerabilities and adding only 6 new ones. Its multi-agent structure appears to mitigate limitations commonly observed in single-agent LLMs, such as difficulties handling partial or incomplete context. However, its inability to determine when to stop editing highlights the need for a human-in-the-loop approach to ensure optimal performance and control.

Table 6.5: Vulnerabilities Removed, Added & Unchanged in C3

Type	Removed	Added	Unchanged
schemafactory-xxe-schema	0	0	1
schemafactory-xxe	0	0	1
active-debug-code-printstacktrace	21	3	0
unencrypted-socket	7	1	0
ecb-cipher	4	0	0
unsafe-reflection	0	0	2
defaulthttpclient-is-deprecated	2	0	0
insecure-hostname-verifier	0	1	0
disallow-old-tls-versions1	0	1	0

6.3 Cross-Cutting Themes and Trade-Offs

While RQ1, RQ2, and RQ3 independently assessed CRANE's performance in terms of vulnerability reduction, semantic alignment, and vulnerability severity, this section synthesizes those findings to reveal broader insights and cross-cutting themes across these evaluation goals.

6.3.1 Trade-Offs Between Security and Semantic Similarity

Interestingly, **no strong or consistent trade-off emerged** between the number of vulnerabilities and semantic similarity. Some configurations (e.g., C3) achieved high security and high alignment, suggesting that, under the right conditions, these goals can be jointly optimized. However, configurations with multiple review iterations (e.g., C6 and C7) improved security (if compared to the Final Dataset) but at the cost of reduced semantic similarity or increased vulnerability severity.

This suggests that while not always in conflict, goals like vulnerability reduction and semantic alignment or vulnerability severity can diverge depending on how CRANE is configured, particularly when layering additional complexity into the review process.

6.3.2 Sensitivity of Configuration Parameters

Among the variables tested, the number of iterations had the most pronounced impact. More iterations resolved less vulnerabilities while introducing new ones, including high-severity issues, and often led to lower semantic similarity. Reviewer personality and vulnerability descriptions offered more consistent benefits with fewer side effects, particularly in C3. Thus, the sensitivity of CRANE to iteration depth suggests that more review is not always better, and automatic pipelines may benefit from controlled stopping conditions or human oversight.

On Trade-Offs

While CRANE's best-performing configuration does not explicitly optimize for a trade-off between security and semantic similarity, the results from the ablation study suggest that such a trade-off may implicitly exist. Enhancements that improve security—such as additional context—can, in some cases, reduce alignment with the original human-written code. This highlights the need for practitioners to prioritize goals based on the specific application context.

6.4 Implications for Practice and Research

Based on our findings:

- **Practitioners** integrating multi-agent LLM frameworks like CRANE into secure coding workflows need to define clear objectives upfront. If the priority is to minimize vulnerabilities, they should adopt configurations enriched with contextual information (e.g., personality traits, detailed vulnerability descriptions), which consistently deliver stronger security outcomes. Conversely, when preserving semantic fidelity to human-written code is more important, simpler configurations with fewer iterations and reviewers will better serve their needs.
- **Researchers** should focus on exploring how different system parameters affect practical trade-offs between security and semantic alignment. The availability of our full replication package [25, 45] enables immediate experimentation with various configurations to tailor CRANE for diverse scenarios.

- **Educators** can leverage these insights to teach students and developers about the balance between security enhancement and maintaining code readability and style, underscoring the role of system configuration and human oversight.

Moreover, this study confirms that human-in-the-loop approaches are critical: manual tuning of CRANE's parameters (number of reviewers, iteration count, contextual detail) significantly impacts performance. Practitioners should therefore incorporate human oversight and iterative adjustment to optimize outcomes in real-world development environments.

Finally, CRANE demonstrates strong potential as a versatile, adaptable multi-agent framework that not only supports developers' decision-making during code generation but also provides explicit, actionable guidance—positioning it as an effective pair-programming assistant that enhances productivity without sacrificing control over the final code.

On Practical Adoption of Multi-Agent LLMs

Successful adoption of multi-agent LLMs like CRANE requires stakeholders to tailor configurations carefully to their goals, recognizing that no single setup fits all use cases. Emphasis on human-in-the-loop strategies and adaptive tuning is essential to harness these tools effectively and responsibly in secure software development.

6.5 Relation to Prior Work

CRANE, designed as a multi-agent LLM framework, offers promising results that suggest such systems—when properly configured—can be effective tools for vulnerability mitigation. Powered by OpenAI's ChatGPT-4o-mini, CRANE supports findings by Noever [38], particularly in its ability to repair security vulnerabilities. It also confirms prior observations regarding LLMs' limitations in determining when to stop editing a code snippet [30].

Interestingly, CRANE appears to challenge other findings from studies such as Yetistiren et al. [47], Husein et al. [49], and Pan et al. [30], which report LLM struggles in low-context environments. In contrast, CRANE demonstrates that, under certain

configurations, GPT-4o-mini can still generate secure code even when provided with only partial context.

Regarding vulnerability types, CRANE largely aligns with existing literature on common weaknesses introduced by LLMs (e.g., CWE-209, CWE-327), though it also surfaces CWE-470—an uncommon reported issue in prior research—indicating a potential gap in existing analyses.

As such, CRANE may serve as a useful baseline for future research on multi-agent LLM-based systems, particularly in secure code generation. Its design and results provide valuable insights into how collaborative agent structures perform when evaluated against real-world datasets.

Finally, the observed performance drop when multiple review iterations are introduced may add empirical weight to theories around self-repair blind spots in LLMs, suggesting that more iterations do not necessarily lead to better security outcomes.

On Relation to Prior Work

CRANE establishes itself as a valid multi-agent framework by both confirming and challenging earlier research: it supports findings on vulnerability detection and types most often introduced by LLMs, while contradicting studies that suggest LLMs cannot operate effectively with limited context. Its limitations and strengths offer a grounded foundation for future work in secure multi-agent LLM systems.

6.6 Limitations

CRANE inherits several limitations from the underlying LLMs, including restricted context windows, execution costs, and performance constraints. These limitations are addressed or mitigated as follows:

- **Short Context Window:** The dataset was filtered to exclude code snippets exceeding 2000 tokens, while the suggestions produced during the Code Review step were summarized.

- **Execution Costs:** To reduce costs, ChatGPT-4o-mini was used instead of the more expensive GPT-4o or GPT-4, balancing performance with affordability.
- **Model Capabilities:** Since CRANE relies solely on ChatGPT-4o-mini, its performance is inherently tied to the strengths and limitations of this specific model. The choice of this model was based on findings by Noever [38], who identified GPT-4 as particularly effective for vulnerability detection and repair tasks.

Despite these limitations, CRANE's modular and flexible architecture makes it adaptable to other domains and programming languages. Because CRANE does not rely on hardcoded assumptions, new tasks can be addressed by adjusting the models' context, and language support is handled dynamically by the LLM. However, as with any LLM-based system, performance may vary depending on the nature of the task and the programming language. CRANE may also inherit biases and blind spots observed in single-agent LLM systems. More informations in Chapter 7.

On CRANE's Limitations

While CRANE inherits common limitations of the underlying LLMs—such as context size, execution cost, and model-specific performance—it remains flexible due to its modular architecture. This makes it adaptable across domains and programming languages. However, any deployment beyond the tested setting should involve re-evaluating configurations and performance assumptions to ensure reliability and task-specific effectiveness.

CHAPTER 7

Threats to validity

This chapter presents the main threats to the validity of the results obtained through the CRANE framework, along with the strategies adopted to mitigate them. Where mitigation was not feasible, residual risks are discussed explicitly.

7.1 Internal Validity

Internal validity refers to the extent to which the results can be confidently attributed to CRANE's system configurations rather than external or confounding factors.

A key threat arises from the non-deterministic nature of LLMs: their outputs can vary across runs due to probabilistic sampling. This randomness may affect both the generated code and the emergent behavior of CRANE's multi-agent system. To control for this, all experiments used fixed random seeds and consistent temperature settings, improving reproducibility and isolating the impact of each configuration.

However, some variability remains possible—especially during deeper iterations involving multiple agents exchanging feedback. Still, this reflects the typical behavior of publicly accessible LLMs and mirrors realistic deployment scenarios.

7.2 External validity

External validity refers to the extent to which the findings of this study can be generalized to different contexts, including other codebases, programming languages, or development environments.

One limitation lies in the exclusive use of Java code from the Gerrit dataset, which may restrict the generalizability of results to other programming languages or domains. However, Java remains a widely used language and is known to present challenges for secure AI-generated code, making it a meaningful and relevant choice for this study. Moreover, the Gerrit dataset provides a rich, real-world corpus of code review scenarios, featuring both secure and vulnerable snippets of varying lengths and complexities.

7.3 Construct Validity

Construct validity relates to whether the metrics and instruments used in the study accurately capture the intended concepts—namely, semantic alignment and security.

To measure semantic similarity, cosine similarity was computed between CRANE's output and human-written code. While it may fail to capture higher-level structural or behavioral alignment, such as algorithmic equivalence or developer intent, it is widely adopted in NLP and code generation research.

The study assessed vulnerability reduction using Semgrep, a rule-based static analysis tool. While Semgrep's limited reasoning capacity can lead to under- or over-reporting of issues, it covers a broad range of vulnerability patterns and is well-suited for integration into CI/CD pipelines. Additionally, Semgrep rules were cross-validated against known CVE-tagged examples to ensure reliability.

A potential limitation is the separate treatment of vulnerability count, semantic similarity, and severity. In practice, trade-offs may occur—for instance, improving security might lower semantic similarity with human-written code. While this interplay is acknowledged, it was not explicitly modeled, allowing a clearer focus on the individual insights offered by each metric.

7.4 Conclusion Validity

Conclusion validity concerns whether the inferences drawn from the data are reasonable and supported by the evidence.

One notable trend observed was that increased review iterations correlated with higher vulnerability severity, particularly in C7. While suggestive, this correlation should not be misinterpreted as causation. The absence of controlled experiments or statistical hypothesis testing limits the ability to assert causality with confidence.

To address this in part, trends were confirmed through qualitative comparisons and repetition across configurations, but deeper statistical analysis remains an opportunity for future work.

CHAPTER 8

Conclusions

8.1 Summary of the Study

The motivation behind this study was to support developers during Secure Code Review (SCR) activities by addressing the limitations often encountered in single-agent systems or unassisted development workflows. To fill this gap, we introduced CRANE, a collaborative multi-agent LLM-based framework designed to assist developers by offering contextual insights, actionable suggestions, and secure code snippets. A key innovation of CRANE lies in its use of demographic persona patterns, which enhance realism and relevance in simulated human-AI collaboration. The framework was developed following the Design Science Research (DSR) methodology, ensuring that the artifact directly responds to problems identified in the literature and contributes to practical and academic knowledge in the field of secure software development.

8.2 Key Findings

When applied to real-world code reviews extracted from Gerrit, CRANE's best-performing configuration achieved an **81.59%** reduction in security vulnerabilities,

while maintaining a strong semantic alignment (cosine similarity of 0.691) with the original code. This shows that secure code can be generated without compromising code intent or meaning.

The significance of this result goes beyond raw metrics. Compared to prior work, for example, the 20% vulnerability reduction reported in Hamer et al. [37], CRANE demonstrated a substantially greater impact. This underscores the practical value of collaborative agent diversity in real-world scenarios.

A key insight emerged from the incorporation of MBTI-based personality modeling: while the improvement was marginal, the presence of cognitive diversity within LLM agents contributed positively to the overall security outcome. This suggests that simulating varied human reasoning styles can enhance system robustness and output quality in collaborative AI configurations.

Ultimately, the findings of this thesis reinforce the core hypothesis: collaborative, multi-agent LLM systems that integrate structured reasoning, behavioral diversity, and secure development practices represent a promising direction for addressing long-standing challenges in AI-assisted code review. CRANE exemplifies how such systems can move from theoretical potential to practical impact.

8.3 Overall Contributions & Impact

This thesis introduces CRANE, a collaborative multi-agent LLM system designed not only to generate code snippets but also to provide meaningful insights that can support both human and AI agents during secure code review tasks. By addressing the limitations commonly associated with single-agent systems—such as incomplete contextual understanding—CRANE offers a practical and adaptable solution for real-world development environments.

CRANE serves as both a collaborator for complex tasks and a substitute for repetitive or lower-level activities, making it a **versatile framework for developers and practitioners**. Its multi-agent architecture—enhanced by human-like cognitive diversity—not only improves task coverage and adaptability but also lays a solid foundation for future research into collaborative and context-aware AI systems.

A key innovation of CRANE lies in its ability to be tailored across two dimensions:

technical specialization, enabled by the demographic persona pattern, and **collaborative behavior**, shaped through the use of MBTI-inspired personality traits. These features make CRANE not only a powerful tool for Secure Code Review but also a novel framework that advances the integration of human-in-the-loop principles in software security workflows.

8.4 Broader Implications and Community Impact

Built in response to limitations highlighted in the literature—such as the high vulnerability rate of AI-generated code and the difficulty of handling context-poor snippets—CRANE demonstrates the potential of collaborative multi-agent LLM systems in secure code review. By incorporating demographic persona patterns and MBTI-driven diversity, CRANE simulates more realistic human interactions and cognitive variety, allowing for both automated and human-in-the-loop usage.

These findings are particularly relevant to the research community and industry practitioners alike. CRANE not only showcases how collaborative LLM configurations can outperform single-agent models but also illustrates how such systems can be effectively deployed in real-world environments. Its integration with Static Analysis Tools (SATs) allows LLMs to go beyond surface-level syntax checks, addressing deeper semantic issues and improving overall code security.

By bridging the gap between traditional static analysis and the reasoning capabilities of LLMs, CRANE sets a new direction for security-aware AI code review. The architecture can be extended or adapted, making it a foundational reference for future tools aiming to support secure and context-sensitive software development.

8.5 Future Work

While CRANE has demonstrated promising results, several avenues remain open for further exploration. One direction involves conducting a more granular ablation study that includes a wider range of parameters—such as the number of exchanged messages between agents—and evaluating performance on a different change request dataset featuring a broader range of CWEs.

Although CRANE was designed as a **human-in-the-loop system**, the current study evaluated it through a fully automated pipeline. Future research could incorporate human agents in the roles of Reviewer, Moderator, or Developer to assess how real-world collaboration influences the effectiveness and efficiency of the framework.

Furthermore, while retrieval-augmented generation (RAG) was primarily used to ensure continuity across iterations, future work could enhance the demographic persona of each agent by enriching them with specialized domain knowledge. This knowledge could be stored in a RAG system, enabling agents to dynamically retrieve task-relevant context based on their assigned specialization.

Lastly, one notable gap is CRANE's performance on other programming languages or larger-scale, real-world projects likely to contain LLM-generated code. In addition, the long-term maintainability of CRANE-generated code has yet to be studied and represents an important direction for future investigation.

Bibliography

[1] R. Shirey, "Internet security glossary," Tech. Rep., 2000. (Cited on page 1)

[2] S. Planning, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, vol. 1, no. 2002, 2002. (Cited on page 1)

[3] 2020, "Gitlab: Mapping the devsecops landscape - 2020 survey." [Online]. Available: https://about.gitlab.com/developer-survey/ (Cited on page 1)

[4] T. W. Thomas, M. Tabassum, B. Chu, and H. Lipford, "Security during application development: An application security expert perspective," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–12. (Cited on page 1)

[5] M. Tahaei and K. Vaniea, "A survey on developer-centred security," in *2019 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*. IEEE, 2019, pp. 129–138. (Cited on page 1)

[6] C. Weir, A. Rashid, and J. Noble, "I'd like to have an argument, please," 2017. (Cited on page 1)

[7] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Software engineering: Barry W. Boehm's lifetime contributions to software development, management, and research*, vol. 34, no. 1, p. 75, 2007. (Cited on page 2)

[8] A. F. Ackerman, P. J. Fowler, and R. G. Ebenau, "Software inspections and the industrial production of software," in *Proc. of a symposium on Software validation: inspection-testing-verification-alternatives*, 1984, pp. 13–40. (Cited on page 2)

[9] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software inspections: an effective verification process," *IEEE software*, vol. 6, no. 3, pp. 31–36, 1989. (Cited on page 2)

[10] A. Bacchelli and C. Bird, "Expectations, outcomes, and challenges of modern code review," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 712–721. (Cited on page 2)

[11] Z. Yang, C. Gao, Z. Guo, Z. Li, K. Liu, X. Xia, and Y. Zhou, "A survey on modern code review: Progresses, challenges and opportunities," *arXiv preprint arXiv:2405.18216*, 2024. (Cited on page 2)

[12] G. McGraw, "Software security," *IEEE Security & Privacy*, vol. 2, no. 2, pp. 80–83, 2004. (Cited on page 2)

[13] C. Thompson and D. Wagner, "A large-scale study of modern code review and security in open source projects," in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering*, 2017, pp. 83–92. (Cited on page 2)

[14] A. Meneely and L. Williams, "Strengthening the empirical analysis of the relationship between linus' law and software security," in *Proceedings of the 2010 ACM-IEEE international symposium on empirical software engineering and measurement*, 2010, pp. 1–10. (Cited on page 2)

[15] A. Meneely and O. Williams, "Interactive churn metrics: socio-technical variants of code churn," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 6, pp. 1–6, 2012. (Cited on page 2)

[16] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, "Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities," *IEEE transactions on software engineering*, vol. 37, no. 6, pp. 772–787, 2010. (Cited on page 2)

[17] A. Poller, L. Kocksch, S. Türpe, F. A. Epp, and K. Kinder-Kurlanda, "Can security become a routine? a study of organizational change in an agile software development group," in *Proceedings of the 2017 ACM conference on computer supported cooperative work and social computing*, 2017, pp. 2489–2503. (Cited on page 2)

[18] S. Turpe, L. Kocksch, and A. Poller, "Penetration tests a turning point in security practices? organizational challenges and implications in a software development {Team}," in *Twelfth Symposium on Usable Privacy and Security (SOUPS 2016)*, 2016. (Cited on page 2)

[19] L. Braz, C. Aeberhard, G. Çalikli, and A. Bacchelli, "Less is more: supporting developers in vulnerability detection during code review," in *Proceedings of the 44th International conference on software engineering*, 2022, pp. 1317–1329. (Cited on pages 2, 3 e 6)

[20] L. Braz and A. Bacchelli, "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*, 2022, pp. 810–821. (Cited on pages 3, 4, 6, 10, 19, 20 e 24)

[21] J. Jiang, F. Wang, J. Shen, S. Kim, and S. Kim, "A survey on large language models for code generation," *arXiv preprint arXiv:2406.00515*, 2024. (Cited on pages 4, 14 e 17)

[22] C. Negri-Ribalta, R. Geraud-Stewart, A. Sergeeva, and G. Lenzini, "A systematic literature review on the impact of ai models on the security of code generation," *Frontiers in Big Data*, vol. 7, p. 1386720, 2024. (Cited on pages 4, 6, 14, 15, 16, 17, 21, 23, 24 e 33)

[23] S. Cruz, F. Q. Da Silva, and L. F. Capretz, "Forty years of research on personality in software engineering: A mapping study," *Computers in Human Behavior*, vol. 46, pp. 94–113, 2015. (Cited on pages 5, 20, 23, 24 e 36)

[24] M. Mukadam, C. Bird, and P. C. Rigby, "Gerrit software code review data from android," in *2013 10th Working Conference on Mining Software Repositories (MSR)*. IEEE, 2013, pp. 45–48. (Cited on pages 6, 20, 24, 29, 33 e 36)

[25] L. Morelli, "Crane crs dataset–online appendix." [Online]. Available: https://doi.org/10.5281/zenodo.15518588 (Cited on pages 6, 24, 40 e 63)

[26] W. Charoenwet, P. Thongtanunam, V.-T. Pham, and C. Treude, "Toward effective secure code reviews: an empirical study of security-related coding weaknesses," *Empirical Software Engineering*, vol. 29, no. 4, p. 88, 2024. (Cited on page 9)

[27] V. Lenarduzzi, F. Pecorelli, N. Saarimaki, S. Lujan, and F. Palomba, "A critical comparison on six static analysis tools: Detection, agreement, and precision," *Journal of Systems and Software*, vol. 198, p. 111575, 2023. (Cited on page 11)

[28] S. Yadav, A. M. Qureshi, A. Kaushik, S. Sharma, R. Loughran, S. Kazhuparambil, A. Shaw, M. Sabry, N. S. J. Lynch, P. O'Hara *et al.*, "From idea to implementation: Evaluating the influence of large language models in software development–an opinion paper," *arXiv preprint arXiv:2503.07450*, 2025. (Cited on pages 12, 13 e 14)

[29] G. Fan, D. Liu, R. Zhang, and L. Pan, "The impact of ai-assisted pair programming on student motivation, programming anxiety, collaborative learning, and programming performance: a comparative study with traditional pair programming and individual approaches," *International Journal of STEM Education*, vol. 12, no. 1, p. 16, 2025. (Cited on pages 12, 13 e 14)

[30] Z. Pan, R. Cao, Y. Cao, Y. Ma, B. Li, F. Huang, H. Liu, and Y. Li, "Codev-bench: How do llms understand developer-centric code completion?" *arXiv preprint arXiv:2410.01353*, 2024. (Cited on pages 14, 17, 60, 61 e 64)

[31] E. BASIC and A. GIARETTA, "From vulnerabilities to remediation: A systematic literature review of llms in code security," 2025. (Cited on pages 14, 17, 23, 24, 33, 56 e 58)

[32] H. Pearce, B. Ahmad, B. Tan, B. Dolan-Gavitt, and R. Karri, "Asleep at the keyboard? assessing the security of github copilot's code contributions," *Communications of the ACM*, vol. 68, no. 2, pp. 96–105, 2025. (Cited on pages 15 e 51)

[33] G. Sandoval, H. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, "Lost at c: A user study on the security implications of large language model code assistants," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 2205–2222. (Cited on page 15)

[34] O. Asare, M. Nagappan, and N. Asokan, "Is github's copilot as bad as humans at introducing vulnerabilities in code?" *Empirical Software Engineering*, vol. 28, no. 6, p. 129, 2023. (Cited on page 15)

[35] Y. Wu, N. Jiang, H. V. Pham, T. Lutellier, J. Davis, L. Tan, P. Babkin, and S. Shah, "How effective are neural networks for fixing security vulnerabilities," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 1282–1294. (Cited on pages 15, 33 e 52)

[36] C. Tony, N. E. D. Ferreyra, and R. Scandariato, "Github considered harmful? analyzing open-source projects for the automatic generation of cryptographic api call sequences," in *2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2022, pp. 896–906. (Cited on page 16)

[37] S. Hamer, M. d'Amorim, and L. Williams, "Just another copy and paste? comparing the security vulnerabilities of chatgpt generated code and stackoverflow answers," in *2024 IEEE Security and Privacy Workshops (SPW)*. IEEE, 2024, pp. 87–94. (Cited on pages 16 e 71)

[38] D. Noever, "Can large language models find and fix vulnerable software?" *arXiv preprint arXiv:2308.10345*, 2023. (Cited on pages 17, 64 e 66)

[39] J. Wang, L. Cao, X. Luo, Z. Zhou, J. Xie, A. Jatowt, and Y. Cai, "Enhancing large language models for secure code generation: A dataset-driven study on vulnerability mitigation," *arXiv preprint arXiv:2310.16263*, 2023. (Cited on pages 17, 21 e 56)

[40] A. R. Hevner, S. T. March, J. Park, and S. Ram, "Design science in information systems research," *MIS quarterly*, pp. 75–105, 2004. (Cited on pages 19, 20, 21 e 22)

[41] E. W. dos Santos and I. Nunes, "Investigating the effectiveness of peer code review in distributed software development based on objective and subjective data," *Journal of Software Engineering Research and Development*, vol. 6, no. 1, p. 14, 2018. (Cited on pages 23, 53 e 54)

[42] M. Jureczko, Ł. Kajda, and P. Górecki, "Code review effectiveness: an empirical study on selected factors influence," *IET Software*, vol. 14, no. 7, pp. 794–805, 2020. (Cited on page 23)

[43] L. F. Capretz and F. Ahmed, "Why do we need personality diversity in software engineering?" *ACM SIGSOFT Software Engineering Notes*, vol. 35, no. 2, pp. 1–11, 2010. (Cited on pages 23, 24 e 36)

[44] A. D. Da Cunha and D. Greathead, "Does personality matter? an analysis of code-review ability," *Communications of the ACM*, vol. 50, no. 5, pp. 109–112, 2007. (Cited on pages 23, 24 e 36)

[45] L. Morelli, "Crane: Code review ai network engine–online appendix." [Online]. Available: https://doi.org/10.5281/zenodo.15518781 (Cited on pages 24, 25, 39, 40 e 63)

[46] Y. Fu, P. Liang, A. Tahir, Z. Li, M. Shahin, J. Yu, and J. Chen, "Security weaknesses of copilot generated code in github," *arXiv preprint arXiv:2310.02059*, 2023. (Cited on pages 51, 56 e 58)

[47] B. Yetiştiren, I. Özsoy, M. Ayerdem, and E. Tüzün, "Evaluating the code quality of ai-assisted code generation tools: An empirical study on github copilot, amazon codewhisperer, and chatgpt," *arXiv preprint arXiv:2304.10778*, 2023. (Cited on pages 52, 54, 60 e 64)

[48] N. Nguyen and S. Nadi, "An empirical evaluation of github copilot's code suggestions," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 1–5. (Cited on pages 53 e 54)

[49] R. A. Husein, H. Aburajouh, and C. Catal, "Large language models for code completion: A systematic literature review," *Computer Standards & Interfaces*, p. 103917, 2024. (Cited on pages 60 e 64)

Acknowledgments

Before switching to Italian, I would like to sincerely thank all the people I met at JADS—individuals truly worth knowing. Among them, a few deserve special recognition. I'm grateful to Federico, Federica, Filippo, Marco and Stefano for all the laughter and the great times we shared; to Renato for being not only a brilliant Dungeon Master, but also someone I could always count on for support. A heartfelt thank you to Cristoffer and Indika—mentors in both professional projects and personal growth, as well as colleagues with whom I could speak openly and honestly. I want to thank Roya for the light-hearted conversations and the contagious smiles that always came with them. Finally, I want to thank Giuseppe and Damian, who closely supported me during this Erasmus experience by suggesting approaches and strategies to improve my work. I learned a lot from you. I truly hope to see you all again. Thank you.

Vorrei ora ringraziare il Prof. Palomba, presenza costante nella mia carriera universitaria da ormai tre anni. Da lei ho imparato molto, crescendo sia a livello personale che professionale. Il suo stile di insegnamento è sempre stato fonte d'ispirazione e, se oggi ho trovato la forza e il coraggio di intraprendere anche io questo percorso, è soprattutto grazie a lei. Nella speranza di future collaborazioni, la ringrazio ancora una volta per la fiducia dimostratami nell'affidarmi questo progetto tanto stimolante quanto complesso.

Desidero inoltre ringraziare Gilberto, Giammaria e Antonio, dottorandi che mi hanno supportato (e sopportato) durante questi mesi. Vi sono grato per i consigli e i

suggerimenti, per i paper "food for thoughts" condivisi e per l'attenzione dimostrata nei momenti in cui mi sono trovato ad affrontare situazioni nuove.

Grazie a David, Adriano e Dario, amici con cui ho condiviso quella che tutt'ora reputo l'esperienza più formativa che abbia mai fatto. Partire per l'Erasmus è sempre stato un mio desiderio, ma non mi sarei mai immaginato di riuscire a convididerlo con persone come voi.

Ringrazio Tommaso, Gerardo, Luigi, Davide e Mario, persone che ho avuto la fortuna di conoscere in questi due anni – e la sfortuna di non aver conosciuto prima. È difficile immaginare la magistrale senza i momenti passati insieme a ridere, scherzare e studiare.

Ringrazio Ndoni e Fdna per essere stati parte integrante dei miei momenti di svago e per esserci stati sin dall'inizio. Mi reputo fortunato a conoscervi.

Ringrazio Marco, Aldo, Filippo, Stefano, Maya e Mereis per aver rappresentato le mie serate di relax, di gioco o di sfogo. Quando tutto si faceva pesante voi c'eravate e, di questo, vi ringrazio.

Vorrei poi ringraziare Daniele. Probabilmente ti meriteresti una pagina intera. Ricordo ancora come se fosse ieri il primo giorno di università, quando, inconsapevole del percorso che mi aspettava, ho conosciuto te. Sono passati tanti anni e, come la prima volta che mi aiutasti per un esame, ti ripeto: senza di te, oggi non sarei qui. Gli esami affrontati insieme, i consigli che mi hai dato, e tutte le volte in cui – invece di darmi la risposta – insistevi perché ci arrivassi da solo, sono ricordi che custodirò sempre gelosamente. Grazie per avermi spronato ad essere migliore. Ti voglio bene.

Grazie a Mattia, Lucrezia e Giuseppe. Ho sempre pensato che passare il tempo con voi riuscisse, anche solo per qualche ora, ad allontanare i problemi che in quel momento mi affliggevano e, di questo, vi ringrazio.

Grazie ad Aurora e Bleffo, membri integranti della mia seconda famiglia. In questi due anni ho avuto la fortuna di approfondire il nostro rapporto, e sarebbe riduttivo considerarvi dei semplici amici. Ci vediamo poco, praticamente una volta l'anno, quando va bene, eppure non serve fare nulla per ritrovare quel clima sereno in cui posso essere semplicemente me stesso, senza filtri.

Vorrei ringraziare i miei nonni, Gino e Ciccio, con i quali non ho avuto la possibilità di condividere questa esperienza, ma che sono certo sarebbero orgogliosi di me.

Vorrei ringraziare le mie nonne, Giovanna e Pepé per essersi interessate al mio percorso universitario, chiedendomi sempre come stessero andando le cose e per supportarmi incondizionatamente.

Vorrei ringraziare Taleesa, la mia sorellina. Ti ho vista crescere in questi anni più forte (e testarda) di me, ma al contempo dolce e gentile. Grazie per avermi sempre augurato buona fortuna prima di un esame, di chiedermi poi come fosse andata o di congraturarti con me per il risultato ottenuto.

Vorrei ringraziare mamma, ormai esperta informatica almeno quanto me. Ti ringrazio per la dispnibilità nel sentirmi blaterare per ore ed ore, per il tuo pieno supporto e per tutte le possibilità datemi.

Vorrei ringraziare papà, sempre pronto a darmi consigli e supporto morale. La mia integrità, il mio essere una persona corretta ed il bene che ho fatto in quanto tale, lo devo a te. Grazie per avermi supportato in qualsiasi scelta.

Infine, vorrei ringraziare Rozzi e Ginevra, membri fondanti della mia seconda famiglia. È difficile anche solo spiegare a parole quanto io sia grato della vostra presena nella mia vita. Sono cresciuto guardando FRIENDS, sognando di avere un gruppo di amici su cui contare ciecamente ma mai e poi mai mi sarei immaginato di poterlo avere davvero. Vorrei concludere dedicandovi il discorso che Dave (in questo caso io) dedica ad Aaron (voi) nel film "The Interview", frasi che descrivono meglio di quanto potrò mai fare ciò che provo per voi: "*Aaron, you are the Sam Wisegamgee to my Frodo Baggins. You are the Gandalf to my Bilbo Baggins. But of all the Lord of the Rings references i could make, this is the most important. I'M GOLLUM and you are my PRECIOUSS. Smeagle needs Aaron.*".