Automated Software Testing using Multi-Agent Systems (MAS) and Large Language Models (LLMs)

Rui Marinho

Student No.: 1171602

A dissertation submitted in partial fulfillment of the requirements for the degree of Master of Science, Specialisation Area of

Supervisor: Constantino Martins

Evaluation Committee:
President:
[Nome do Presidente, Categoria, Escola]

Members:
[Nome do Vogal1, Categoria, Escola]
[Nome do Vogal2, Categoria, Escola] (até 4 vogais)

Porto, January 14, 2026

Statement Of Integrity

[Maintain only the version corresponding to the main language of the work]

I hereby declare that I have conducted this academic work with integrity.

I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

Therefore, the work presented in this document is original and was authored by me, having not been previously used for any other purpose. The exceptions [REMOVE THIS CLAUSE IF IT DOES NOT APPLY - REMOVE THIS COMMENT] are explicitly acknowledged in the section that addresses ethical considerations. This section also states how Artificial Intelligence tools were used and for what purpose.

I further declare that I have fully acknowledged the Code of Ethical Conduct of P.PORTO.

ISEP, Porto, [Month] [Day], [Year]

Declaro ter conduzido este trabalho académico com integridade.

Não plagiei ou apliquei qualquer forma de uso indevido de informações ou falsificação de resultados ao longo do processo que levou à sua elaboração.

Portanto, o trabalho apresentado neste documento é original e de minha autoria, não tendo sido utilizado anteriormente para nenhum outro fim. As exceções [REMOVER ESTE PERÍODO NO CASO DE NÃO SE APLICAR - APAGAR ESTE COMENTÁRIO] estão explicitamente reconhecidas na secção onde são abordadas as considerações éticas. Esta secção também declara como as ferramentas de Inteligência Artificial foram utilizadas e para que finalidade.

Declaro ainda que tenho pleno conhecimento do Código de Conduta Ética do P.PORTO.

ISEP, Porto, [Dia] de [Mês] de [Ano]

Dedicatory

The dedicatory is optional. Below is an example of a humorous dedication.

"To my wife Marganit and my children Ella Rose and Daniel Adam without whom this book would have been completed two years earlier." in "An Introduction To Algebraic Topology" by Joseph J. Rotman.

Abstract

This document explains the main formatting rules to apply to a Master Dissertation work for the MSc in Artificial Intelligence Engineering of the Computer Engineering Department (DEI) of the School of Engineering (ISEP) of the Polytechnic of Porto (IPP).

The rules here presented are a set of recommended good practices for formatting the disseration work. Please note that this document does not have definite hard rules, and the discussion of these and other aspects of the development of the work should be discussed with the respective supervisor(s).

This document is based on a previous document prepared by Dr. Fátima Rodrigues (DEI/ISEP).

The abstract should usually not exceed 200 words, or one page. When the work is written in Portuguese, it should have an abstract in English.

Please define up to 6 keywords that better describe your work, in the *THESIS INFORMATION* block of the `main.tex` file.

Keywords: Keyword1, ..., Keyword6

Resumo

Após o resumo/abstract é obrigatório colocar as principais palavras-chave/keywords do tema em que se insere o trabalho desenvolvido, sendo permitido um máximo de 6 palavras-chave/keywords, estas devem ser caraterizadoras do trabalho desenvolvido e surgirem com frequência no documento escrito.

Para alterar a língua basta ir às configurações do documento no ficheiro `main.tex` e alterar para a língua desejada ('english' ou 'portuguese')[1]. Isto fará com que os cabeçalhos incluídos no template sejam traduzidos para a respetiva língua.

Palavras-chave: Keyword1, ..., Keyword6

[1] Alterar a língua requer apagar alguns ficheiros temporários; O target **clean** do **Makefile** incluído pode ser utilizado para este propósito.

Acknowledgement

The optional Acknowledgment goes here... Below is an example of a humorous acknowledgment.

"I'd also like to thank the Van Allen belts for protecting us from the harmful solar wind, and the earth for being just the right distance from the sun for being conducive to life, and for the ability for water atoms to clump so efficiently, for pretty much the same reason. Finally, I'd like to thank every single one of my forebears for surviving long enough in this hostile world to procreate. Without any one of you, this book would not have been possible." in "The Woman Who Died a Lot" by Jasper Fforde.

Contents

List of Algorithms xix

List of Source Code xxi

1 Introduction 1
- 1.1 Contextualization . . . 1
  - 1.1.1 The Economic Imperative of Automation . . . 2
  - 1.1.2 From Static Analysis to Generative Reasoning . . . 2
- 1.2 Problem Definition . . . 2
- 1.3 Hypothesis . . . 3
- 1.4 General Objective . . . 3
- 1.5 Specific Objectives . . . 3
- 1.6 Expected Results and Contributions . . . 4
  - 1.6.1 Scientific Contributions . . . 4
  - 1.6.2 Technical Contributions . . . 4
  - 1.6.3 Expected Results . . . 4
- 1.7 Document Structure . . . 5

2 State of the Art 7
- 2.1 Introduction . . . 7
- 2.2 Methodology . . . 7
  - 2.2.1 Research Questions . . . 8
  - 2.2.2 Search Strategy . . . 8
    - Data Sources . . . 8
    - Search Strings . . . 8
  - 2.2.3 Inclusion and Exclusion Criteria . . . 9
  - 2.2.4 Study Selection Process . . . 9
  - 2.2.5 Quality Assessment . . . 10
- 2.3 Synthesis of Findings . . . 10
  - 2.3.1 RQ1: Architecture & Orchestration . . . 10
    - Hierarchical Orchestration (Waterfall) . . . 10
    - Cooperative Orchestration (Feedback Loops) . . . 11
    - Self-Reflection and Debugging . . . 11
  - 2.3.2 RQ2: Knowledge Integration . . . 11
    - Retrieval-Augmented Generation (RAG) . . . 11
    - Agent-Computer Interfaces (ACI) . . . 11
  - 2.3.3 RQ3: Validation & Evaluation . . . 12
    - Benchmarks: From HumanEval to SWE-bench . . . 12
    - LLM-as-a-Judge . . . 12
- 2.4 Discussion . . . 12
- 2.5 Ethical Considerations . . . 13

2.5.1 Data Privacy and Intellectual Property . . . 13
2.5.2 Bias and Fairness in Testing . . . 13
2.5.3 Impact on the Workforce . . . 13
2.5.4 Energy Consumption and Sustainability . . . 13
2.6 Conclusion . . . 14

3 Methodology and Tools **15**
3.1 Introduction . . . 15
3.2 Methodological Approach . . . 15
3.2.1 Architectural Design . . . 15
3.3 Tools and Technologies . . . 16
3.3.1 Letta Framework . . . 16
3.3.2 Supporting Technologies . . . 16
3.4 Data Collection and Experimental Evaluation . . . 16
3.4.1 Data Collection Strategy . . . 16
3.4.2 Experimental Design . . . 17
3.4.3 Evaluation Metrics . . . 17
3.5 Ethical and Security Implementation . . . 17
3.6 Conclusion . . . 17

4 Conclusions and Future Work **19**
4.1 Summary of the Work . . . 19
4.2 Discussion of Contributions . . . 19
4.3 Future Work . . . 20
4.4 Final Remarks . . . 20

Bibliography **21**

A Appendix Title Here **23**

List of Figures

2.1 PRISMA 2020 Flow Diagram for the Systematic Literature Review. 10

List of Tables

2.1 Research Questions 8
2.2 Selected Data Sources 8
2.3 Search Strings Strategy 9
2.4 Inclusion and Exclusion Criteria 9

List of Algorithms

List of Source Code

List of Symbols

a	distance	m
P	power	W ($\mathrm{J\,s^{-1}}$)
ω	angular frequency	rad

Chapter 1

Introduction

This chapter provides a comprehensive overview of the research context, defining the problem scope and the motivation behind using Multi-Agent Systems for software testing. It outlines the central hypothesis guiding the work, the specific research objectives, and the expected scientific and technical contributions of this dissertation.

1.1 Contextualization

Software testing stands as one of the most resource-intensive yet indispensable phases in the software development lifecycle (SDLC). It is the primary mechanism for ensuring system reliability, security, and adherence to user requirements. In the context of modern Continuous Integration/Continuous Deployment (CI/CD) pipelines, the demand for rapid, automated testing has never been higher. Winters (2020) in "Software Engineering at Google" emphasize that as systems scale, the linearity of manual testing becomes a bottleneck that halts development velocity. Consequently, the industry has traversed a long evolutionary path: from purely manual verification to script-based automation frameworks like Selenium and JUnit, and now, towards the era of Artificial Intelligence (AI).

Historically, test automation was synonymous with writing code to test code. Frameworks such as JUnit for Java or PyTest for Python allowed developers to codify assertions. While this represented a significant leap over manual clicking, it introduced the "maintenance trap." As the application code evolved, the rigid test scripts would break, requiring constant human intervention to update selectors, logic, and data mocks. This fragility led to the search for more resilient, adaptive testing methods.

The introduction of the Transformer architecture by Vaswani (2017) marked a watershed moment. Large Language Models (LLMs) trained on vast repositories of source code (e.g., GitHub, StackOverflow) demonstrated an emergent ability to understand not just natural language, but the syntax and semantics of programming languages. Models like Codex (powering GitHub Copilot) and later GPT-4 proved capable of generating unit tests, documenting legacy code, and even translating between languages. This capability promised to alleviate the burden of test writing, theoretically allowing developers to generate comprehensive test suites from simple natural language prompts.

However, the initial excitement around "Generative AI for Code" faced a reality check when applied to complex, enterprise-grade systems. A single interaction with an LLM (a "prompt") is inherently stateless and limited by its context window. It struggles to hold the architecture of a million-line repository in its "working memory." To address this, the field is shifting towards Multi-Agent Systems (MAS). In this paradigm, the "AI" is not a single chatbot but

a team of specialized agents—a "Product Manager" agent that breaks down requirements, a "Developer" agent that writes the code, a "QA" agent that reviews it, and a "Tester" agent that attempts to break it. Frameworks like MetaGPT (Hong 2024) and ChatDev (Qian 2024) illustrate this collaborative approach, showing that agents with distinct personas and feedback loops can solve problems that overwhelm a single model.

1.1.1 The Economic Imperative of Automation

The cost of software defects rises exponentially the later they are discovered in the development lifecycle. A bug found during the requirements phase costs a fraction to fix compared to one discovered in production, which may incur reputational damage, data loss, and significant engineering hours for remediation. Traditional test automation aims to "shift left," moving testing earlier in the cycle. However, the creation of robust test suites is itself an expensive engineering endeavor. Estimates suggest that for every hour of feature development, up to 0.5 to 1 hour is spent writing and maintaining tests. In large organizations, this translates to millions of dollars annually spent on test maintenance rather than innovation. The promise of Autonomous Software Testing (AST) powered by agents is not merely technical but economic: decoupling test coverage from human effort.

1.1.2 From Static Analysis to Generative Reasoning

Before LLMs, "automated" testing often meant Static Application Security Testing (SAST) or fuzzing. Tools like SonarQube or AFL (American Fuzzy Lop) are powerful but limited. SAST looks for known patterns (e.g., SQL injection vulnerabilities) but cannot understand business logic. Fuzzing throws random data at inputs to find crashes but cannot reason about *why* a function exists. LLMs bridge this gap by bringing "semantic understanding." An LLM can read a function named `calculate_mortgage_interest`, understand that interest cannot be negative, and generate a test case specifically checking for negative input. This semantic reasoning capability distinguishes GenAI-based testing from all previous generations of tools.

1.2 Problem Definition

Despite the promise of agentic AI, the automated generation of reliable, executable test cases for enterprise software remains a complex and unsolved engineering challenge. The core problem lies in the disconnect between the generative capabilities of Large Language Models (LLMs) and the strict correctness requirements of software execution environments. While LLMs excel at pattern matching and generating syntactically plausible code, they lack an inherent understanding of the specific runtime constraints, internal dependencies, and business logic of proprietary codebases.

This disconnect manifests as a "Grounding Gap": the model operates in a probabilistic text space, while the compiler operates in a deterministic logic space. A single hallucinated method call or incorrect import renders an entire test suite useless. Furthermore, existing tools often treat test generation as a one-off "fire-and-forget" task, failing to mimic the human engineering process of writing, executing, analyzing error logs, and iteratively refining the code. The absence of this feedback loop prevents autonomous agents from self-correcting, leading to high-maintenance test artifacts that require significant human intervention to function.

The central problem addressed by this dissertation is the inability of current single-agent Large Language Model approaches to autonomously generate correct, executable, and high-coverage test suites for complex enterprise software systems. This failure stems from three specific deficiencies:

1. The Oracle Problem: Single-prompt models cannot reliably determine the "correct" expected behavior of code without execution, leading to tests that assert incorrect values or hallucinate non-existent functionality.
2. Contextual Blindness: Models lack access to the broader repository context (e.g., file structure, installed libraries, custom utilities), resulting in generated code that fails to compile due to missing dependencies or incorrect paths.
3. Open-Loop Generation: Current systems lack a mechanism for iterative refinement based on compiler and runtime feedback, preventing them from correcting simple syntax errors or logic bugs that a human developer would fix immediately.

1.3 Hypothesis

This research is driven by the following central hypothesis:

> "A Multi-Agent System, specifically one composed of specialized roles with access to an execution environment and iterative feedback loops, will significantly outperform single-prompt Large Language Models in the generation of valid, executable, and high-coverage test cases for complex software repositories."

1.4 General Objective

The primary goal of this research is to design, implement, and evaluate a Multi-Agent System (MAS) Framework that orchestrates specialized autonomous agents to generate, validate, and refine software tests. The framework aims to bridge the "Grounding Gap" by integrating Agent-Computer Interfaces (ACI) that allow agents to interact with real execution environments, thereby achieving higher rates of functional correctness and code coverage than single-agent baselines.

1.5 Specific Objectives

To achieve the general objective, the following specific objectives (SOs) are defined:

- SO1: Define a taxonomy of specialized agent roles (e.g., Planner, Coder, Tester, Reviewer) and their interaction protocols to mimic a collaborative engineering workflow.
- SO2: Design and implement an Agent-Computer Interface (ACI) that provides agents with safe, controlled access to external tools, including file system navigation, static linters, and test runners (e.g., PyTest).
- SO3: Develop a "Self-Healing" feedback loop mechanism that enables agents to parse execution error logs (stderr) and iteratively refine their generated code to resolve compilation and logic errors.

- SO4: Empirically evaluate the proposed framework using standard industry benchmarks (e.g., SWE-bench), measuring key performance indicators such as Pass Rate (Pass@1), Code Coverage percentage, and the reduction in human intervention required.

1.6 Expected Results and Contributions

The contributions of this dissertation are multifaceted, addressing both the scientific advancement of Automated Software Engineering and the practical needs of the software industry.

1.6.1 Scientific Contributions

This research contributes to the scientific body of knowledge by:

- **Taxonomy of Agentic Roles:** Defining a rigorous classification of agent responsibilities in the testing domain, moving beyond generic "chatbots" to role-specific prompting strategies (e.g., the specific cognitive load of a "Test Designer" vs. a "QA Auditor").
- **Feedback Loop Dynamics:** Providing empirical evidence on how compiler and runtime feedback signals influence the convergence rate of LLM-generated code. This helps quantify the value of "Grounding" in probabilistic generation.
- **Ethical Framework for Autonomous Development:** Offering a structured analysis of the ethical implications of replacing human verification with agentic consensus, contributing to the debate on "Human-in-the-Loop" governance.

1.6.2 Technical Contributions

From a technical and informatic perspective, the dissertation delivers:

- **MAS Framework Implementation:** A reusable, modular framework enabling the orchestration of multiple LLM agents (using tools like Letta or LangGraph) to autonomously navigate a repository and generate tests.
- **Agent-Computer Interface (ACI) Design:** A concrete implementation of an ACI tailored for software testing, including safe sandboxing of agent-executed code and structured error parsing.
- **Self-Healing Pipeline:** A robust CI/CD-compatible pipeline where agents act as "maintenance bots," automatically attempting to fix broken tests before alerting human developers.

1.6.3 Expected Results

The expected outcomes of this research include:

- A fully functional prototype capable of generating unit tests for Python repositories with a compilation success rate exceeding 80% (compared to <50% for zero-shot baselines).
- A comprehensive benchmark dataset comparing the proposed MAS approach against standard baselines (e.g., GPT-4o, GitHub Copilot) on the SWE-bench Light dataset.

- A reduction in the "Human Effort" metric, measured by the number of manual edits required to make a generated test suite passable.

1.7 Document Structure

The remainder of this document is organized as follows:

- Chapter 2: State of the Art provides a systematic literature review, analyzing the evolution of MAS in testing, current architectural patterns, and validation methodologies.
- Chapter 3: Methodology & Tools details the proposed system architecture, the selection of the Letta framework, and the design of the experimental setup.
- Chapter 4: Experimentation [Future Work] will present the results of the empirical evaluation and the analysis of the collected data.
- Chapter 5: Conclusion [Future Work] will interpret the findings, discuss limitations, and outline future research directions.

Chapter 2

State of the Art

This chapter presents a Systematic Literature Review (SLR) of the current landscape of Multi-Agent Systems (MAS) in software engineering. It details the methodology used for selecting studies, synthesizes findings regarding agent architectures and knowledge integration, and discusses the ethical implications of deploying autonomous agents in development workflows.

2.1 Introduction

The rapid advancement of Large Language Models (LLMs) has precipitated a paradigm shift in Software Engineering (SE), particularly in the domain of automated software testing. While traditional automated testing relies heavily on static analysis and manually scripted test cases, the emergence of Generative AI (GenAI) offers the potential for autonomous, context-aware test generation. However, single-prompt LLM interactions often fail to address the complexity of enterprise-grade software due to hallucinations, limited context windows, and a lack of grounding in the execution environment. Consequently, the research frontier has moved towards Multi-Agent Systems (MAS), where specialized agents collaborate to plan, generate, execute, and refine tests.

This chapter presents a Systematic Literature Review (SLR) conducted to rigorously analyze the current state of the art in MAS-driven automated testing. Following the PRISMA 2020 guidelines (Page 2021), this review systematically identifies, selects, and synthesizes 25 primary studies published between 2023 and 2025. The goal is to answer critical questions regarding the architectural orchestration of these agents, the integration of domain-specific knowledge, and the validation methodologies used to ensure reliability. Furthermore, this chapter includes a dedicated analysis of the ethical, legal, and environmental implications of deploying autonomous agents in software development workflows.

2.2 Methodology

To ensure transparency, reproducibility, and scientific rigor, this SLR adopts a structured methodology based on the PRISMA (Preferred Reporting Items for Systematic Reviews and Meta-Analyses) framework. The review process was executed in four distinct phases: (1) Definition of Research Questions, (2) Search Strategy, (3) Study Selection, and (4) Data Extraction and Synthesis.

2.2.1 Research Questions

The primary objective of this review is to understand how MAS architectures can overcome the limitations of monolithic LLMs in software testing. To this end, three specific Research Questions (RQs) were formulated, as detailed in Table 2.1.

Table 2.1: Research Questions

ID	Research Question
RQ1	Architecture & Orchestration: How are specialized agents within Multi-Agent Systems architecturally decomposed, coordinated, and orchestrated to accomplish complex software testing tasks?
RQ2	Knowledge Integration: What methodologies (e.g., Retrieval-Augmented Generation, Tool Use, Agent-Computer Interfaces) are employed to integrate proprietary, domain-specific knowledge into LLM-based test generation systems?
RQ3	Validation & Evaluation: How do existing studies evaluate the correctness, coverage, and effectiveness of LLM-generated test scripts, and what benchmarks are considered the gold standard?

2.2.2 Search Strategy

Data Sources

Given the rapid pace of development in Generative AI, the search was conducted across the following major academic databases (see Table 2.2).

Table 2.2: Selected Data Sources

Database	Justification
Semantic Scholar	To identify citation networks and relevant peer-reviewed papers in venues such as ICSE, FSE, and ASE.
IEEE Xplore	To ensure coverage of formally published archival literature in engineering and computer science.
ACM Digital Library	To capture high-impact proceedings from major computing conferences.

Search Strings

To ensure a comprehensive retrieval of relevant studies, a multi-string search strategy was employed. Instead of a single broad query, three distinct boolean search strings were constructed to target specific dimensions of the research questions, as detailed in Table 2.3.

The search was conducted in December 2025, covering the period from January 2023 to December 2025. This timeframe was selected to focus specifically on the "post-ChatGPT" era, where agentic capabilities became viable.

Table 2.3: Search Strings Strategy

ID	Scope	Search String
S1	Core Scope	("Software Testing" OR "Test Generation" OR "Unit Testing" OR "Fuzzing") AND ("Multi-Agent" OR "MAS" OR "Agentic" OR "Autonomous Agents") AND ("LLM" OR "Large Language Model")
S2	Validation & Benchmarking	("Software Testing") AND ("LLM" OR "Agent") AND ("Benchmark" OR "SWE-bench" OR "HumanEval" OR "Pass@k" OR "Metric")
S3	Ethics & Privacy	("Software Engineering") AND ("LLM" OR "Agent") AND ("Privacy" OR "GDPR" OR "Data Leakage" OR "Bias" OR "Energy")

2.2.3 Inclusion and Exclusion Criteria

The selection process was governed by rigorous inclusion and exclusion criteria to ensure the relevance and quality of the selected studies, as defined in Table 2.4.

Table 2.4: Inclusion and Exclusion Criteria

ID	Criterion
IC1	Population: Software development environments, focusing on code repositories and testing workflows.
IC2	Intervention: Multi-Agent Systems (MAS) utilizing LLMs as the reasoning engine.
IC3	Outcome: Quantitative metrics such as Pass@k, Code Coverage, or Qualitative assessments.
EC1	Papers focused solely on single-prompt engineering without agentic loops.
EC2	Studies lacking empirical validation or reproducible benchmarks.
EC3	Non-English publications.
EC4	Grey literature not backed by technical reports.

2.2.4 Study Selection Process

The systematic search process yielded a total of 87 records across the selected databases. After removing duplicates (12), 75 unique citations were screened based on title and abstract. This initial screening led to the exclusion of 25 records that did not meet the population or intervention criteria (e.g., general NLP papers). The remaining 50 full-text articles were assessed for eligibility. Of these, 25 were excluded for reasons such as lack of empirical validation (EC2) or focus on single-agent prompting (EC1). The final set comprises 25 primary studies that form the basis of the synthesis presented in Section 2.3. The selection flow is illustrated in Figure 2.1.

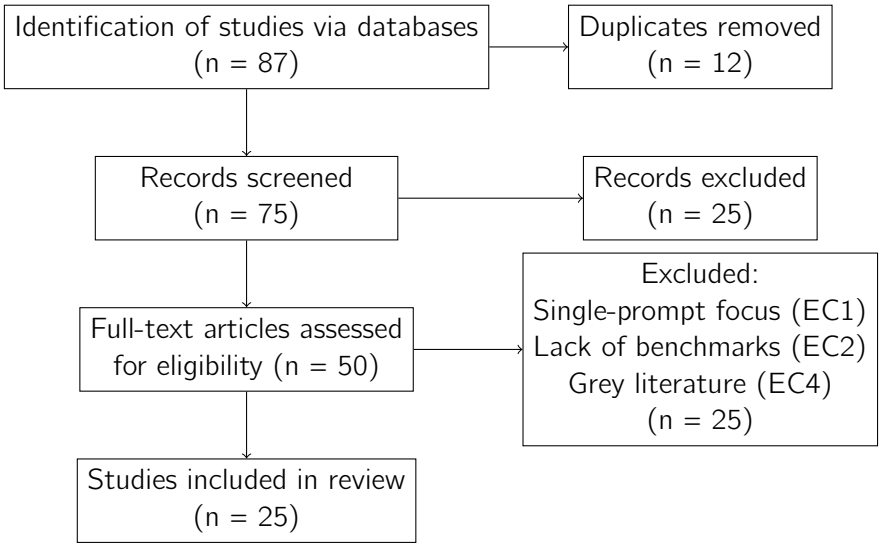

Figure 2.1: PRISMA 2020 Flow Diagram for the Systematic Literature Review.

2.2.5 Quality Assessment

Each selected study was evaluated for quality based on three factors: (1) Reproducibility (availability of code/datasets), (2) Benchmarking (use of standard benchmarks like SWE-bench vs. ad-hoc datasets), and (3) Architectural Clarity (clear definition of agent roles and communication protocols).

2.3 Synthesis of Findings

This section synthesizes the findings from the selected primary studies, structured according to the three research questions defined in the methodology. It begins by analyzing the architectural patterns of MAS, explores the mechanisms for integrating domain knowledge, and concludes with an assessment of validation strategies.

2.3.1 RQ1: Architecture & Orchestration

The literature reveals a decisive move from single-agent systems to multi-agent architectures. The synthesis identifies two primary architectural patterns: Hierarchical and Cooperative orchestration.

Hierarchical Orchestration (Waterfall)

In hierarchical systems, agents are organized in a top-down structure mimicking a corporate hierarchy. Hong (2024) introduced MetaGPT, which assigns roles such as Product Manager, Architect, Project Manager, and Engineer. The workflow follows a strict Standard Operating Procedure (SOP), where the output of one agent (e.g., a PRD from the Manager) serves as the immutable input for the next (e.g., a System Design from the Architect). This structure reduces "drift" and ensures that code generation is aligned with high-level requirements. However, it can be rigid, struggling with tasks that require iterative "back-and-forth" debugging.

Cooperative Orchestration (Feedback Loops)

Cooperative architectures focus on iterative refinement through peer review. Qian (2024) proposed ChatDev, where agents (e.g., Coder and Reviewer) engage in a dialogue to discuss implementation details before writing code. Huang (2024) further specialized this for testing with AgentCoder. In this framework, a "Test Designer" agent generates test cases, which are then used to verify the code produced by a "Programmer" agent. If tests fail, the feedback is fed back to the Programmer in a loop. This multi-agent feedback loop was found to significantly outperform single-pass generation, increasing Pass@1 rates on the HumanEval benchmark by over 15%.

Self-Reflection and Debugging

Recent works like CodeCoR (Pan 2025) and RGD (Jin 2024) emphasize "Self-Reflection." Here, agents do not just generate code but also generate a rationale or analysis of their own errors. For example, LDB (Large Language Model Debugger) generates a "bug report" before attempting a fix, separating the fault localization step from the patch generation step. This decomposition of the testing task—Plan, Generate, Test, Fix—is the hallmark of modern MAS architectures.

2.3.2 RQ2: Knowledge Integration

A critical limitation of off-the-shelf LLMs is their lack of knowledge about specific enterprise codebases. The review identifies two dominant strategies for knowledge integration: Retrieval-Augmented Generation (RAG) and Agent-Computer Interfaces (ACI).

Retrieval-Augmented Generation (RAG)

RAG allows agents to query a vector database containing the project's documentation and source code. Chen (2025) explored how RAG aids code generation, finding that retrieving relevant API documentation significantly reduces hallucination of non-existent methods. However, naive RAG (retrieving top-k chunks) often fails for testing because test generation requires understanding the logic of the code, not just keyword matches. Advanced techniques use "Graph-based RAG," where the retrieval follows the Call Graph or Control Flow Graph (CFG) of the application.

Agent-Computer Interfaces (ACI)

Perhaps the most significant innovation is the concept of the ACI, formalized by Yang (2024) in SWE-agent. Just as humans use IDEs, agents need an interface to interact with the OS. An ACI provides agents with tools to:

- Search: ‘grep‘ or ‘find‘ files in a repository.
- Read: View file contents with line numbers.
- Edit: Apply patches to specific line ranges.
- Execute: Run ‘pytest‘ or ‘make‘ and capture the ‘stdout‘/‘stderr‘.

This "grounding" is crucial. Alshahwan (2024) at Meta demonstrated with TestGen-LLM that agents grounded in the execution environment (i.e., those that can run the tests they

generate) achieve a higher fix rate for regressions than those that operate in a "blind" text-generation mode. The ability to see the error message allows the agent to iteratively repair the test case.

2.3.3 RQ3: Validation & Evaluation

Validating the output of generative models is notoriously difficult. The SLR highlights a transition from static similarity metrics (e.g., BLEU score) to execution-based functional correctness.

Benchmarks: From HumanEval to SWE-bench

Early studies relied on HumanEval or MBPP (Mostly Basic Python Problems), which consist of self-contained algorithmic puzzles. However, Jimenez et al. (2024) argued that these do not represent real-world software engineering. This led to the creation of SWE-bench, a dataset of real GitHub issues and pull requests from popular libraries (e.g., Django, scikit-learn). Badertdinov (2025) introduced SWE-rebench to ensure evaluation rigor, showing that many agents that perform well on HumanEval fail catastrophically on SWE-bench due to the complexity of file dependencies and environment setup.

LLM-as-a-Judge

Running full test suites is computationally expensive. To mitigate this, Mündler (2024) and others propose "LLM-as-a-Judge". In this paradigm, a strong model (e.g., GPT-4) evaluates the quality of test cases generated by a weaker/cheaper model (e.g., Llama-3-8B). The judge checks for:

- Readability: Does the test follow naming conventions?
- Coverage: Does the test target the edge cases implied by the requirements?
- Logic: Does the test assertion make sense?

While not a replacement for execution, LLM-as-a-Judge provides a rapid, scalable feedback signal during the generation phase.

2.4 Discussion

The analysis of the state of the art reveals a clear trajectory in automated software testing: the move from static, brittle scripts to dynamic, agentic workflows. However, this transition is not without its challenges. The primary problem identified in the literature is the reliability of agentic code generation in large-scale environments. While agents like ChatDev perform well on greenfield projects, they struggle with "Contextual Blindness" when modifying existing brownfield repositories.

This dissertation aims to solve this specific problem by implementing an enhanced ACI that provides agents with "spatial awareness" of the codebase—allowing them to map dependencies before attempting modifications. Unlike existing approaches that rely on naive RAG, our proposed solution will integrate a semantic code graph to guide the agent's navigation. Furthermore, the "Self-Healing" mechanism proposed addresses the open-loop limitation identified in Section 2.3, ensuring that the system can autonomously recover from the compilation errors that currently plague single-shot generations. By closing the loop between

generation and execution, we expect to achieve a significant improvement in the Pass@1 metric for real-world issues.

2.5 Ethical Considerations

The deployment of autonomous agents in software testing introduces novel ethical challenges that extend beyond technical correctness. As these systems become more capable, it is imperative to establish a governance framework that ensures safety, fairness, and accountability.

2.5.1 Data Privacy and Intellectual Property

A paramount concern in enterprise adoption is the exposure of Intellectual Property (IP). Sending proprietary source code to external LLM providers (e.g., OpenAI, Anthropic) via API raises significant risks of data leakage. Even if providers promise not to train on API data, the transmission itself may violate strict data residency laws such as the GDPR (European Union 2024). Techniques like PII Masking and the use of locally hosted open-weights models (e.g., Llama 3, Mixtral) are discussed as necessary mitigations (Nunez 2024).

2.5.2 Bias and Fairness in Testing

LLMs are trained on public code, which reflects the biases of the open-source community. If training data is dominated by certain coding styles or frameworks, agents may exhibit testing bias. For example, an agent might rigorously test "happy paths" commonly found in tutorials but neglect edge cases relevant to accessibility or diverse user inputs. Ugarte (2025) highlight that automated safety testing must explicitly account for these biases to prevent the deployment of discriminatory software. The Association for Computing Machinery (2018) emphasizes the responsibility of software professionals to design systems that are inclusive and accessible; ensuring agents share these values is an open research challenge.

2.5.3 Impact on the Workforce

The automation of QA roles raises fears of job displacement. However, the literature suggests a transformation rather than an elimination of roles. The role of the "QA Engineer" is evolving into that of an "Agent Auditor". Humans will increasingly be responsible for defining the high-level testing strategy, configuring the agentic workflows, and reviewing the most critical failures, while agents handle the high-volume generation of unit and regression tests. This shift requires upskilling in AI literacy and system design.

2.5.4 Energy Consumption and Sustainability

Finally, the environmental impact of MAS cannot be ignored. A single complex task in frameworks like MetaGPT can trigger dozens of API calls and generate thousands of tokens. The energy cost of these inference chains is orders of magnitude higher than running a static analysis tool. Sustainable AI practices, such as caching RAG results and using smaller, specialized models for routine tasks, are emerging as a critical area of research to ensure the ecological viability of agentic SE.

2.6 Conclusion

This Systematic Literature Review confirms that the integration of Multi-Agent Systems with Large Language Models represents a transformative leap in automated software testing. By decomposing tasks, integrating external tools via ACIs, and employing rigorous execution-based validation, MAS architectures address the key limitations of hallucinations and lack of context that plagued earlier single-agent approaches. However, significant challenges regarding data privacy, legacy language support, and energy efficiency must be addressed before widespread enterprise adoption is feasible. The findings of this review directly inform the design of the framework proposed in this dissertation.

Chapter 3

Methodology and Tools

This chapter outlines the methodological approach adopted for the development of the Multi-Agent System (MAS) framework. It details the architectural design, the selection of specific tools and technologies (including the Letta framework), and the strategies for data collection and experimental validation.

3.1 Introduction

The complexity of orchestrating autonomous agents requires a robust engineering foundation. This chapter transitions from the theoretical exploration of the state of the art to the practical implementation details of the proposed system. It describes the "Planner-Actor-Critic" architecture used to coordinate agent activities and justifies the selection of the technology stack. Furthermore, it defines the experimental protocols that will be used to gather data and evaluate the system's performance against the objectives defined in Chapter 1.

3.2 Methodological Approach

The research follows a Design Science Research (DSR) methodology. DSR is chosen because the primary goal is to create a novel artifact (the MAS Framework) that solves a specific problem (automated test generation). The process involves iterative cycles of design, development, and evaluation.

3.2.1 Architectural Design

The system architecture is built upon a modular MAS pattern. It distinguishes between the "Control Plane" (where agents plan and communicate) and the "Data Plane" (where code is executed and tested).

- **Planner Agent:** Responsible for analyzing the repository structure and creating a test strategy.
- **Coder Agent:** Generates the actual test code based on the plan.
- **Executor Agent:** Runs the tests in a sandboxed environment and captures output.

3.3 Tools and Technologies

The implementation relies on a modern stack of AI and software engineering tools. A key component of this stack is the Letta framework, which serves as the backbone for agent state management.

3.3.1 Letta Framework

Letta (formerly associated with MemGPT) is utilized to manage the context and memory of the agents. Unlike stateless LLM calls, Letta allows agents to maintain a persistent "working memory" of the codebase they are analyzing. This is critical for solving the "Contextual Blindness" problem identified in Chapter 2.

Letta organizes memory into two distinct tiers:

- **Core Memory:** Stores the agent's persona (e.g., "You are a Senior QA Engineer") and human instructions. This memory is always present in the context window.
- **Archival Memory:** A larger storage area (backed by a vector database) where the agent can offload information about files, previous test runs, and documentation. The agent can retrieve this information on-demand, allowing it to work with repositories that exceed the LLM's context limit.

3.3.2 Supporting Technologies

In addition to Letta, the following tools are integrated:

- **LLM Backend:** OpenAI GPT-4o and open-weights models (Llama 3) serve as the reasoning engines.
- **Vector Database:** ChromaDB is used for RAG operations, allowing semantic search over the codebase.
- **Containerization:** Docker is used to sandbox the execution environment, preventing generated code from affecting the host system.

3.4 Data Collection and Experimental Evaluation

This section details the plan for data collection ("Recolha de Dados") and the experimental methods ("Experimentação") that will be used to validate the proposed solution.

3.4.1 Data Collection Strategy

The project relies on the **SWE-bench Light** dataset for evaluation. This dataset is a curated collection of real-world GitHub issues (bug reports and feature requests) paired with the pull requests that resolved them.

- **Source:** The dataset includes repositories like Django, scikit-learn, and flask.
- **Selection Criteria:** A random sample of 50 issues will be selected to serve as the test bed.

- **Ground Truth:** Each issue comes with a "Gold Patch" and a set of "Fail-to-Pass" tests. The agent's goal is to autonomously generate a test that reproduces the bug (fails on the original code) and passes on the fixed code.

3.4.2 Experimental Design

The evaluation will compare the proposed MAS framework against two baselines:

1. **Baseline A (Zero-Shot):** A single GPT-4o agent prompted to "write a test for this issue" without access to tools or memory.
2. **Baseline B (RAG-only):** A single agent with RAG access but no iterative feedback loop.

3.4.3 Evaluation Metrics

The success of the solution will be measured using the following key performance indicators (KPIs):

- **Pass@1:** The percentage of issues for which the agent generates a passing test suite on the first attempt.
- **Code Coverage:** The percentage of lines in the target module covered by the generated tests.
- **Self-Correction Rate:** The frequency with which the agent successfully fixes a failing test after analyzing the error log (demonstrating the efficacy of the feedback loop).

3.5 Ethical and Security Implementation

Aligning with the ethical considerations discussed in Section 2.5, this project implements specific safeguards to ensure data protection and safety during the experimental phase.

- **Sandboxing:** All agent-generated code is executed within isolated Docker containers with no network access to the outside world, preventing accidental execution of malicious code.
- **Data Privacy:** The framework is designed to scrub Personally Identifiable Information (PII) from logs before sending prompts to external LLM providers.
- **Human-in-the-Loop:** The system includes a "break-glass" mechanism where a human operator can intervene and halt the agent loop if it detects deviations from the safety policy.

3.6 Conclusion

This chapter has established the blueprint for the implementation phase. By leveraging the state-management capabilities of Letta and adhering to a rigorous DSR methodology, the proposed framework is positioned to address the limitations of existing solutions. The next steps involve the full implementation of the agentic loops and the execution of the pilot experiments.

Chapter 4

Conclusions and Future Work

This chapter summarizes the work developed in the context of this dissertation proposal, highlighting the preliminary findings from the literature review and the expected impact of the proposed Multi-Agent System (MAS). It also outlines the future work required to complete the dissertation, including the implementation roadmap and experimental validation.

4.1 Summary of the Work

The automation of software testing has long been a goal of the software engineering community. While previous generations of tools relied on static analysis and brittle scripts, the advent of Large Language Models (LLMs) has opened new avenues for "generative testing." However, as identified in Chapter 2, current single-agent approaches suffer from significant limitations, most notably "Contextual Blindness" (lack of repository awareness) and the "Grounding Gap" (inability to verify code correctness).

This dissertation proposes a novel solution to these problems: a Multi-Agent System framework grounded in the execution environment. By utilizing the Letta framework for state management and implementing an Agent-Computer Interface (ACI), the proposed system mimics the workflow of a human engineering team. The "Planner" agent strategies, the "Coder" agent implements, and the "Executor" agent validates—creating a self-correcting feedback loop that is absent in standard "Chat with Code" interfaces.

4.2 Discussion of Contributions

The anticipated contributions of this work are both scientific and technical. Scientifically, it will provide empirical evidence regarding the efficacy of agentic feedback loops, potentially quantifying the value of "agent collaboration" over "prompt engineering." Technically, the delivery of a reusable, Docker-sandboxed framework for autonomous testing represents a practical tool that can be adopted by the industry to reduce the "maintenance burden" of legacy codebases.

Furthermore, the ethical framework developed in Section 2.5 and implemented in Section 3.4 ensures that this automation does not come at the cost of safety or privacy. By embedding PII scrubbing and human-in-the-loop safeguards, the project addresses the growing concerns regarding the deployment of autonomous AI in enterprise environments.

4.3 Future Work

To achieve the objectives outlined in Chapter 1, the following tasks remain:

- **Framework Implementation (Months 1-2):** Finalize the integration of Letta with the Dockerized execution environment. Implement the specific prompts for the Planner and Reviewer personas.
- **Data Collection (Month 3):** Execute the baseline experiments using GPT-4o on the SWE-bench Light dataset to establish a performance floor.
- **Evaluation (Month 4):** Run the full MAS framework on the same dataset and analyze the results. Focus on the "Self-Correction Rate" to validate the hypothesis that feedback loops improve code quality.
- **Dissertation Writing (Month 5):** Compile the results into the final dissertation document, refining the discussion based on the empirical data.

4.4 Final Remarks

This dissertation proposal addresses a critical gap in the current state of AI for Software Engineering. By moving beyond simple text generation to agentic orchestration, it aims to unlock the true potential of LLMs as reliable, autonomous partners in the software development lifecycle. The work done so far—defining the problem, reviewing the state of the art, and architecting the solution—provides a solid foundation for the successful execution of the project.

Bibliography

Alshahwan, Nadia et al. (2024). "Automated Unit Test Improvement using Large Language Models at Meta". In: *International Conference on Software Engineering (ICSE)*.

Association for Computing Machinery (2018). *ACM Code of Ethics and Professional Conduct*. url: `https://www.acm.org/code-of-ethics`.

Badertdinov, Ibragim et al. (2025). "SWE-rebench: An Automated Pipeline for Task Collection and Decontaminated Evaluation of Software Engineering Agents". In.

Chen, Jingyi et al. (2025). "When LLMs Meet API Documentation: Can Retrieval Augmentation Aid Code Generation Just as It Helps Developers?" In.

European Union (2024). *Regulation (EU) 2024/1689 of the European Parliament and of the Council of 13 June 2024 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)*. url: `http://data.europa.eu/eli/reg/2024/1689/oj`.

Hong, Sirui et al. (2024). "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework". In: *International Conference on Learning Representations (ICLR)*.

Huang, Dong et al. (2024). "AgentCoder: Multi-Agent-based Code Generation with Iterative Testing and Optimisation". In: *International Conference on Software Engineering (ICSE)*.

Jimenez, Carlos E et al. (2024). "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *The Twelfth International Conference on Learning Representations*.

Jin, Haolin et al. (2024). "RGD: Multi-LLM Based Agent Debugger via Refinement and Generation Guidance". In.

Mündler, Niels et al. (2024). "Code Agents are State of the Art Software Testers". In.

Nunez, Ana et al. (2024). "AutoSafeCoder: A Multi-Agent Framework for Securing LLM Code Generation through Static Analysis and Fuzz Testing". In.

Page, Matthew J et al. (2021). "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* 372.

Pan, Ruwei et al. (2025). "CodeCoR: An LLM-Based Self-Reflective Multi-Agent Framework for Code Generation". In.

Qian, Chen et al. (2024). "ChatDev: Communicative Agents for Software Development". In: *Annual Meeting of the Association for Computational Linguistics (ACL)*.

Ugarte, Miriam et al. (2025). "ASTRAL: Automated Safety Testing of Large Language Models". In.

Vaswani, Ashish et al. (2017). "Attention is all you need". In.

Winters, Titus et al. (2020). *Software Engineering at Google: Lessons Learned from Programming Over Time*.

Yang, John et al. (2024). "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *Conference on Neural Information Processing Systems (NeurIPS)*.

Appendix A

Appendix Title Here

Write your Appendix content here.