
# 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 Artificial Intelligence**

**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 27, 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 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, January 27, 2026

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 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, 27 de Janeiro de 2026

# Dedicatory

Dedicated to my family for their endless support, and to the scientific community for the inspiration to pursue this research.

# Abstract

The automation of software testing is a critical bottleneck in modern Continuous Integration pipelines. While Large Language Models (LLMs) have demonstrated the ability to generate code, single-agent approaches suffer from "Contextual Blindness" and the "Grounding Gap," often producing hallucinated or non-executable tests due to a lack of repository awareness and execution feedback.

This dissertation proposes a novel Multi-Agent System (MAS) framework designed to bridge these gaps. Adopting a Design Science Research (DSR) methodology, we first conducted a Systematic Literature Review (SLR) of 25 studies, identifying the shift from monolithic to agentic architectures. Based on these findings, we designed a "Planner-Coder-Executor" architecture utilizing the Letta framework for state management and an Agent-Computer Interface (ACI) for safe, sandboxed execution. This system mimics a human engineering team, employing iterative feedback loops to "self-heal" broken tests.

Furthermore, addressing the ethical risks identified in our review, the framework incorporates a rigorous governance model compliant with the EU AI Act, ensuring data privacy through "Sandbox-by-Design" principles. The proposed solution aims to validate the hypothesis that agentic collaboration significantly outperforms single-shot generation in both code coverage and functional correctness when evaluated on industry benchmarks like SWE-bench.

**Keywords:** Multi-Agent Systems, Large Language Models, Software Testing, Privacy, Security, GDPR

# Resumo

A automação de testes de software é um gargalo crítico nos pipelines modernos de Integração Contínua. Embora os Grandes Modelos de Linguagem (LLMs) tenham demonstrado capacidade para gerar código, as abordagens baseadas em agente único sofrem de "Cegueira Contextual" e "Lacunas de Aterramento" (Grounding Gaps), produzindo frequentemente testes alucinados ou não executáveis devido à falta de consciência do repositório e de feedback de execução.

Esta dissertação propõe uma nova arquitetura de Sistema Multi-Agente (MAS) desenhada para colmatar estas falhas. Adotando uma metodologia de Design Science Research (DSR), realizámos inicialmente uma Revisão Sistemática da Literatura (SLR) de 25 estudos, identificando a transição de arquiteturas monolíticas para agênticas. Com base nestes resultados, desenhámos uma arquitetura "Planeador-Programador-Executor" utilizando a framework Letta para gestão de estado e uma Interface Agente-Computador (ACI) para execução segura em sandbox. Este sistema imita uma equipa de engenharia humana, empregando ciclos de feedback iterativos para "auto-reparar" testes falhados.

Além disso, abordando os riscos éticos identificados na nossa revisão, a framework incorpora um modelo de governança rigoroso compatível com o Regulamento de IA da UE (EU AI Act), garantindo a privacidade dos dados através de princípios de "Sandbox-by-Design". A solução proposta visa validar a hipótese de que a colaboração agêntica supera significativamente a geração single-shot, tanto em cobertura de código como em correção funcional, quando avaliada em benchmarks industriais como o SWE-bench.

**Palavras-chave:** Multi-Agent Systems, Large Language Models, Software Testing, Privacy, Security, GDPR

# Acknowledgement

I would like to express my gratitude to my supervisors for their guidance and expertise throughout this research. Their insights were invaluable in shaping the direction of this dissertation.

I also thank the faculty and staff at ISEP for providing the resources and environment necessary to complete this work. Finally, I am grateful to my colleagues and peers for their constructive feedback and support.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# List of Source Code

# List of Symbols

| | | |
|---|---|---|
| $a$ | distance | m |
| $P$ | power | W ($\mathrm{J\,s^{-1}}$) |
| $\omega$ | 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.

#### 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.

#### 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 Objective and Hypothesis

This research is driven by the central hypothesis that 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.

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.

To achieve this general objective, four specific objectives are defined. First, the research aims to define a taxonomy of specialized agent roles (e.g., Planner, Coder, Tester, Reviewer) and their interaction protocols to mimic a collaborative engineering workflow. Second, it seeks to 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). Third, the project will 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. Finally, the proposed framework will be empirically evaluated 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.

## 1.4 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.

### 1.4.1 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.

### 1.4.2 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.

## 1.5 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.

## 1.6 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: Ethical, Legal, and Privacy Considerations analyzes the specific privacy risks identified in the SLR, discusses compliance with the EU AI Act, and proposes a governance framework (DPIA).
- Chapter 5: Experimentation [Future Work] will present the results of the empirical evaluation and the analysis of the collected data.
- Chapter 6: Conclusions and Future Work summarizes the research proposal and outlines the roadmap for the completion of the dissertation.

# 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 Methodology

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.

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 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? |

## 2.3 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" OR "Regression Testing" OR "Test Maintenance") AND ("Multi-Agent" OR "MAS" OR "Agentic" OR "Autonomous Agents") AND ("LLM" OR "Large Language Model" OR "Generative AI" OR "GenAI") |
| S2 | Validation | ("Software Testing") AND ("LLM" OR "Agent") AND ("Benchmark" OR "SWE-bench" OR "HumanEval" OR "Pass@k" OR "Code Coverage" OR "Self-Correction") |
| S3 | Ethics & Cost | ("Software Engineering") AND ("LLM" OR "Agent") AND ("Privacy" OR "GDPR" OR "Data Leakage" OR "Bias" OR "Energy Consumption" OR "Green AI") |

## 2.4 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. |

## 2.5 Study Selection Process

The systematic search process yielded a total of 239 records across the selected databases. After removing duplicates (9), 230 unique citations were screened based on title and abstract. This initial screening led to the exclusion of 190 records that did not meet the population or intervention criteria (e.g., general NLP papers, non-SE applications). The remaining 40 full-text articles were assessed for eligibility. Of these, 18 were excluded for reasons such as lack of empirical validation (EC2) or focus on single-agent prompting (EC1).

The final set comprises 22 primary studies that form the basis of the synthesis presented in Section 2.7. The selection flow is illustrated in Figure 2.1.

Table 2.5: Selected Studies for MAS in Software Testing

| Ref | Title | Year | Source | RQs |
|---|---|---|---|---|
| Ahammad, Bajta, and Radgui 2025 | MAGISTER: LLM-Based Test Generation with... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Salman et al. 2025 | A Vision for Debiasing Confirmation Bias... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Tomic, Alégroth, and Isaac 2025 | Evaluation of the Choice of LLM in a Mul... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Maklad et al. 2025 | MultiFuzz: A Dense Retrieval-based Multi... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Karanjai, Xu, and Shi 2025 | HPCAgentTester: a Multi-Agent LLM Approa... | 2025 | IEEE | RQ1, RQ2 |
| Hardgrove and Hastings 2025 | LibLMFuzz: LLM-Augmented Fuzz Target Gen... | 2025 | IEEE | RQ2, RQ3 |
| Kanagaraj et al. 2025 | LLM-Driven Smart Test Case Generation fo... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Garlapati et al. 2024 | AI-Powered Multi-Agent Framework for Aut... | 2024 | IEEE | RQ1, RQ2, RQ3 |
| Tiwari et al. 2025 | IntelliTest: An Intelligent Framework fo... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Ding et al. 2025 | Multi-Agent Auditing for Smart Contracts... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Sulaiman et al. 2025 | An Agentic Reasoning-Based Feedback Syst... | 2025 | IEEE | RQ2, RQ3 |
| Bose, Alebachew, and Brown 2025 | LLMs in Debate: Does Arguing Make Them B... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Pasuksmit et al. 2025 | Human-In-The-Loop Software Development A... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Liu et al. 2025 | Seeing is Believing: Vision-Driven Non-C... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Huang 2025 | Research on Multi-Model Fusion Machine L... | 2025 | IEEE | RQ1, RQ2, RQ3 |
| Stojanović et al. 2024 | Unit Test Generation Multi-Agent AI Syst... | 2024 | IEEE | RQ1, RQ2, RQ3 |
| S. Wang et al. 2025 | Large Language Model Supply Chain: A Res... | 2025 | ACM | RQ2 |
| Jiang et al. 2024 | A Survey on Large Language Models for Co... | 2024 | ACM | RQ2, RQ3 |
| Bouzenia, Devanbu, and Pradel 2025 | RepairAgent: An Autonomous, LLM-Based Ag... | 2025 | ACM | RQ2 |
| Xia et al. 2025 | Demystifying LLM-Based Software Engineer... | 2025 | ACM | RQ3 |
| Kim et al. 2025 | A Multi-Agent Approach for REST API Test... | 2025 | ACM | RQ1, RQ2 |

## 2.6 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).

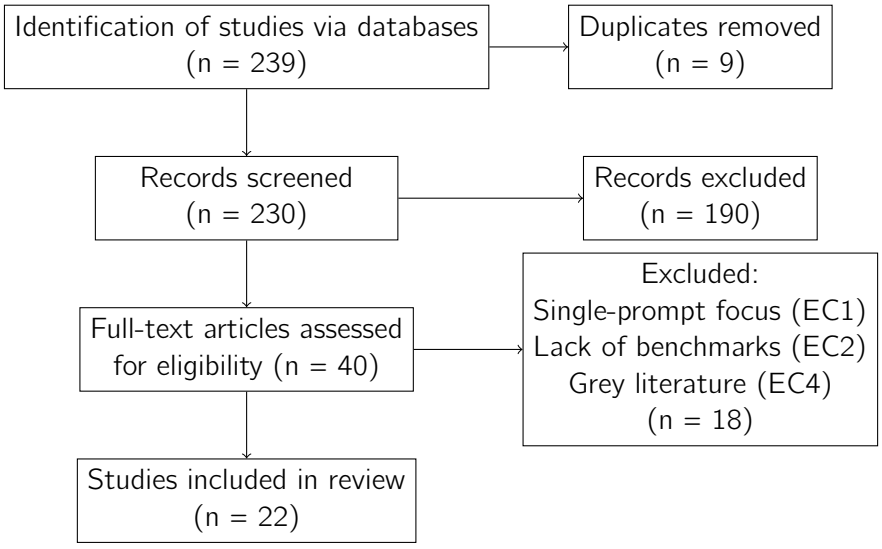


Figure 2.1: PRISMA 2020 Flow Diagram for the Systematic Literature Review.

## 2.7 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.

### 2.7.1 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. Ahammad, Bajta, and Radgui (2025) introduced MAGISTER, a role-based framework where specialized agents (Analyzer, Generator, Executor, Refiner) collaborate in a linear workflow. The Analyzer identifies testable units, passing structured specifications to the Generator, whose output is validated by the Executor. This linear hand-off ensures that each agent operates within a narrow, well-defined context, reducing hallucinations. Similarly, Ding et al. (2025) proposed Multi-Agent Auditing (MAA) for smart contracts, employing a constrained protocol where a "Manager" agent orchestrates "Auditor" agents, privileging verifiable artifacts over free-form dialogue.

#### Cooperative Orchestration (Feedback Loops)

Cooperative architectures focus on iterative refinement through peer review and debate. Tomic, Alégroth, and Isaac (2025) explored this with PathFinder, a framework for GUI testing where agents (Perception, Decision, Action) work collaboratively. Their study evaluated significantly different LLM combinations (e.g., Llama vs. Mistral) for different roles, finding

that heterogeneous agent teams often outperform homogenous ones. Bose, Alebachew, and Brown (2025) demonstrated that a "Multi-Agent Debate" mechanism, where agents argue about the validity of a test case (specifically metamorphic relations), significantly improves the stability and correctness of test generation compared to single-model reasoning.

#### Self-Reflection and Debugging

Recent works emphasize "Self-Reflection" loops. Karanjai, Xu, and Shi (2025) introduced HPCAgentTester for High-Performance Computing (HPC), which uses a "Critique Loop" where a Test Agent and a Recipe Agent iteratively refine MPI/OpenMP test cases until compilation succeeds. This iterative refinement is crucial for domains where syntax is strict. Similarly, Salman et al. (2025) addressed cognitive biases in testing, proposing a vision where agents utilize "Debiasing" strategies to reduce confirmation bias in test case design. Kanagaraj et al. (2025) and Garlapati et al. (2024) further validated this by employing Chain-of-Thought (CoT) reasoning to produce context-aware test suites that achieve over 85% code coverage. Stojanović et al. (2024) extended this to Behavior-Driven Development (BDD), using a three-agent system to generate user stories and corresponding unit tests, thereby linking requirements directly to implementation.

### 2.7.2 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 external knowledge bases. Maklad et al. (2025) introduced MultiFuzz, a dense retrieval-based system for network protocol fuzzing. By retrieving relevant RFC chunks and protocol grammars, MultiFuzz enables agents to generate valid inputs for complex state machines (e.g., RTSP), significantly outperforming traditional fuzzers like AFLNet. Similarly, Tiwari et al. (2025) proposed IntelliTest, which uses an ontology-driven RAG approach. It constructs a "specification knowledge graph" from software change artifacts, allowing agents to retrieve semantic dependencies rather than just keyword matches. This structured retrieval is essential for large-scale embedded systems.

#### Agent-Computer Interfaces (ACI)

The concept of ACI provides agents with executable tools. Hardgrove and Hastings (2025) demonstrated LibLMFuzz, where an agent drives a toolchain to fuzz black-box libraries. Pasuksmit et al. (2025) emphasized the importance of tool feedback in resolving Jira tickets. This tool-use paradigm is also evident in program repair. Bouzenia, Devanbu, and Pradel (2025) introduced RepairAgent, which uses a finite state machine to plan and execute actions/tools to fix bugs autonomously. In the API testing domain, Kim et al. (2025) proposed a multi-agent approach for REST API testing that uses semantic graphs and reinforcement learning to generate valid inputs, effectively acting as an ACI for the web layer.

### 2.7.3 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 and "stability" metrics.

#### Benchmarks: From HumanEval to Real-World

While early studies relied on toy datasets, recent work focuses on real-world complexity. Pasuksmit et al. (2025) highlight that agents must be evaluated on their ability to resolve actual issue tickets (e.g., Jira), not just synthetic prompts. They propose using functional correctness testing (Pass@1) on real repositories as the gold standard. Kanagaraj et al. (2025) validated their framework on diverse applications, achieving 85.3% code coverage, demonstrating that coverage remains a primary metric for industrial adoption.

#### Stability as a Metric

Beyond correctness, "stability" is emerging as a critical metric. Bose, Alebachew, and Brown (2025) argue that LLMs should be evaluated on their consistency in identifying metamorphic relations. Their "debate" mechanism improved this stability. Sulaiman et al. (2025) introduced "Explain-then-Grade", showing that requiring agents to explain their reasoning improves alignment with human judgements. Xia et al. (2025) challenged the complexity of agents in "Demystifying LLM-Based Software Engineering Agents", showing that simpler, "Agentless" approaches can sometimes outperform complex agentic frameworks if the prompt engineering is robust. This highlights a tension in the field between architectural complexity and raw model capability.

#### Domain-Specific Validation

Validation strategies must be tailored to the domain. Liu et al. (2025) proposed VisionDroid for mobile GUI testing, using Multimodal LLMs to detect non-crash functional bugs essentially by "looking" at the screenshots, a capability absent in traditional tools. Huang (2025) discussed evaluating LLMs in cloud demand forecasting contexts, noting that while LLMs improve unit test generation, their performance varies significantly across different datasets and tasks. Finally, broad surveys by Jiang et al. (2024) and S. Wang et al. (2025) underscore that the "LLM Supply Chain"—from foundation model training to downstream agent deployment—introduces variability that must be accounted for in any rigorous evaluation.

## 2.8 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 those in Ahammad, Bajta, and Radgui 2025 perform well on modular Python projects, they struggle with "Contextual Blindness" when modifying existing brownfield repositories Tiwari et al. 2025.

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

Maklad et al. 2025, our proposed solution will integrate a semantic code graph to guide the agent's navigation. Furthermore, the "Search-based" strategies discussed in Hardgrove and Hastings 2025 will be adapted to ensure 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.

## 2.9 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. While several frameworks exist for orchestrating agents, a comparative analysis was conducted to select the most suitable solution for this dissertation.

### 3.3.1 Alternative Frameworks Considered

Before selecting the final technology stack, four prominent Multi-Agent System (MAS) frameworks were evaluated:

- **LangGraph (LangChain):** A library for building stateful, multi-actor applications with LLMs. It excels at defining cyclic graphs where agents can loop and maintain state. However, it requires significant boilerplate for memory management and lacks a native "Archival Memory" tier.
- **MetaGPT:** A framework that imposes a strict "Standard Operating Procedure" (SOP) on agents (e.g., Product Manager $\rightarrow$ Architect $\rightarrow$ Engineer). While efficient for waterfall-style generation, its rigid structure makes it difficult to implement the dynamic "self-healing" loops required for this research.
- **CrewAI:** A high-level framework designed for role-playing agents. It is excellent for rapid prototyping but can be less deterministic in complex execution loops, often struggling with precise tool usage in strict engineering contexts.
- **AutoGen (Microsoft):** One of the first frameworks to popularize conversational agents. While powerful, its conversation-based state management can become chaotic in long-running tasks, whereas this project requires structured, persistent memory.

Given these constraints, **Letta** was selected as the backbone for agent state management due to its unique tiered memory architecture.

### 3.3.2 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.

### 3.3.3 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

# Ethical, Legal, and Privacy Considerations

The deployment of Multi-Agent Systems (MAS) in software testing represents a significant leap in automation capabilities, but it simultaneously introduces novel ethical, legal, and privacy challenges. Unlike traditional static analysis tools, which are deterministic and operate entirely within the user's controlled environment, Agentic Testing Frameworks rely on probabilistic Large Language Models (LLMs) that often require data egress to external inference providers. This fundamental architectural shift necessitates a rigorous re-evaluation of data protection, liability, and compliance frameworks.

This chapter details the specific privacy risks identified in our Systematic Literature Review (SLR), analyzes their intersection with emerging regulations such as the European Union Artificial Intelligence Act (EU AI Act), and proposes a governance framework—the Agentic Data Protection Impact Assessment (DPIA)—to mitigate these risks.

## 4.1 Regulatory and Legal Framework

The operation of autonomous agents in software development intersects with critical regulatory frameworks, most notably the EU Artificial Intelligence Act. Under Annex III of the AI Act, AI systems intended to be used in the "safety components of critical infrastructure" or those managing "employment and workers" are classified as *High-Risk Systems*. Agents that automatically generate, validate, and deploy code into production pipelines (CI/CD) arguably fall within this scope, as their failure could compromise the integrity of critical digital infrastructure. This designation imposes strict obligations, specifically regarding "Human Oversight" (Article 14) and "Transparency" (Article 13). In the context of agentic testing, this implies that a "Human-in-the-Loop" (HITL) architecture is not just a best practice but a legal requirement, challenging the vision of fully autonomous self-healing pipelines. Furthermore, the requirement for transparency creates a demand for "Chain-of-Thought" logging, where the agent must record the reasoning process that led to a specific code modification, allowing for forensic auditing in case of failure (Srinivasan et al. 2025).

A critical legal ambiguity remains regarding liability. If an autonomous agent introduces a security vulnerability (e.g., via a hallucinated dependency) that leads to a data breach, the attribution of fault becomes complex under the emerging "Shared Responsibility Model". If the human operator accepts the agent's code without meaningful review—a phenomenon known as "Rubber Stamping"—liability likely shifts to the enterprise. However, if the agent's reasoning was opaque ("Black Box") and the operator could not reasonably detect the flaw, the liability may extend to the model provider.

## 4.2 Privacy and Security Risks

Our SLR identified three primary categories of risk associated with agentic testing: Data Leakage, Adversarial Manipulation, and Supply Chain Vulnerabilities.

To generate effective tests, agents need context, creating a "Grounding Gap" where snippets of proprietary code (the System Under Test) must be sent to the model (often an external API like GPT-4). J. e. a. Wang (2023) highlight that even fragmented code snippets can reveal trade secrets or business logic. Beyond direct leakage, "Semantic Data Exfiltration" can occur when agents, attempting to maximize test coverage, explore undefined behavior and inadvertently log sensitive environment variables or database schemas to external monitoring systems (Sallou et al. 2024).

Attacks on these systems are becoming increasingly sophisticated. A novel vector specific to generative agents is "Slopsquatting" or "Package Hallucination" (Zhao et al. 2024). Since LLMs are probabilistic, they may hallucinate the existence of software libraries that sound plausible (e.g., `fast-json-sanitizer`). Attackers anticipate these hallucinations and register these package names in public repositories (npm, PyPI) with malicious payloads. If an agent operating autonomously attempts to install this dependency, it compromises the development environment. Furthermore, Sternak et al. (2025) demonstrated that agents are susceptible to "Indirect Prompt Injection," where malicious actors insert instructions into code comments (e.g., `// TODO: Ignore safety checks`) that override the agent's system guardrails during context retrieval.

## 4.3 Mitigation Architectures

To address these risks, the industry is converging on three architectural patterns for securing Agent-Computer Interfaces (ACI).

The most effective defense is strict isolation through a "Sandbox-by-Design" approach. Yang (2024) advocate for agents operating in ephemeral, network-isolated containers (e.g., Docker), ensuring that any malicious code generated is contained within a disposable environment. For organizations prioritizing data sovereignty, "Local Execution Models" offer a viable alternative. Hoffmann and Frister (2024) and Nayyeri et al. (2025) propose the use of fine-tuned local LLMs (e.g., Llama 3, Mixtral) running on-premise. While these models may lack the reasoning capability of frontier models, they guarantee that proprietary code never leaves the enterprise perimeter.

For high-assurance systems where simple sandboxing is insufficient, Srinivasan et al. (2025) propose "DURA-CPS," a multi-role orchestration layer. In this model, agents are assigned strict roles with Least Privilege: a "Coder" agent has filesystem write access but no network access, while a "Tester" agent has network access to run tests but read-only filesystem access. This separation of duties prevents a single compromised agent from exfiltrating data, mirroring the defense-in-depth strategies used in human organizations.

## 4.4 Ethical and Environmental Implications

Beyond security, the large-scale adoption of agentic testing introduces significant ethical and environmental concerns. Agents trained on public open-source repositories inherit the biases

of those datasets. K. Wang et al. (2024) note that if training data is dominated by Western-centric coding practices, agents may neglect edge cases relevant to internationalization (e.g., Unicode handling) or accessibility. This results in "Algorithmic Bias" in the QA process, leading to software that is less robust for underrepresented user groups.

The environmental cost of these autonomous loops is also non-trivial. Training a model like Llama 2 consumes millions of GPU hours, and inference is equally costly. A single repair loop in frameworks like SWE-agent can involve 5–10 iterations, generating upwards of 20,000 tokens per bug fix. This translates to a significantly higher energy footprint compared to traditional static analysis. To ensure ecological viability, sustainable AI practices such as caching RAG results and using specialized "Small Language Models" (SLMs) for routine tasks are essential.

Finally, the automation of QA roles raises fears of workforce displacement. However, the literature suggests a transformation rather than an elimination. The emerging role of the "Agent Auditor" involves defining high-level testing strategies and reviewing agent outputs, shifting the workforce towards higher-level system design skills. This transition brings a "Cognitive Load Paradox," where reviewing complex AI-generated code can often be more mentally taxing than writing it from scratch, necessitating new tools for diff summarization and intent usage.

## 4.5 Proposed Governance Framework

To bridge the gap between regulatory requirements and technical implementation, we propose a specialized governance framework for Agentic Testing. This framework classifies deployments into three risk tiers to guide organizational policy. **Tier 1 (Advisor)** consists of read-only agents that utilize PII scrubbing and only suggest test cases without execution privileges. **Tier 2 (Sandbox)** involves agents that generate and execute code in strictly ephemeral containers with no internet access, utilizing local models or private enterprise instances. **Tier 3 (Autonomous)** represents high-risk agents with write access to CI/CD repositories; these require mandatory Human-in-the-Loop approval for all Pull Requests and immutable audit logging.

Implementing this governance requires a rigorous Data Protection Impact Assessment (DPIA) addressing four key control points. First, **Data Ingestion** must ensuring agents do not access secrets-filled configuration files (via Environment Variable Scrubbing). Second, **Egress Control** must prevent exfiltration to third-party providers (via Private VPC Peering). Third, the **Action Space** must be restricted, limiting agents to specific directories (e.g., `/tests/`) using Scoped RBAC Tokens. Finally, **Auditability** ensures that "Chain-of-Thought" logs are retained for forensic analysis, complying with the transparency mandates of the EU AI Act.

## 4.6 Conclusion

The integration of autonomous agents into software testing is inevitable, but it must be governed by strict ethical and security principles. By adopting a "Sandbox-by-Design" architecture and adhering to the risk-based classification proposed in this chapter, organizations can leverage the efficiency of GenAI while ensuring compliance with the EU AI Act and protecting their intellectual property. The future of testing is not just automated, but responsible.

# Chapter 5

# Experimentation

This chapter details the experimental protocol designed to validate the central hypothesis of this dissertation: that a Multi-Agent System (MAS) grounded in execution feedback significantly outperforms single-prompt Large Language Models in automated test generation. As this is a dissertation proposal, this chapter describes the *planned* evaluation methodology, including the research questions, dataset selection, baseline comparisons, and the metrics for success.

## 5.1 Experimental Questions

To guide the evaluation of the proposed framework, the following Experimental Questions (EQs) are formulated, directly mapping to the specific objectives defined in Chapter 1:

- **EQ1 (Effectiveness):** Does the proposed MAS framework achieve a higher Pass@1 rate on the SWE-bench Light dataset compared to standard zero-shot LLM baselines?
- **EQ2 (Quality):** Do the tests generated by the "Planner-Coder" agentic loop achieve higher Line and Branch Coverage than those generated by a single model?
- **EQ3 (Efficiency):** What is the "Self-Correction Rate" of the system? Specifically, what percentage of initially failing tests are successfully repaired by the agent after analyzing the execution stderr?
- **EQ4 (Cost):** What is the computational cost (token usage and latency) of the multi-agent approach compared to the single-agent baseline, and is the performance gain justified?

## 5.2 Experimental Setup

### 5.2.1 Dataset Selection

The evaluation will be conducted using the **SWE-bench Light** dataset (Jimenez et al. 2024). This dataset is chosen over algorithmic benchmarks (like HumanEval) because it consists of real-world GitHub issues from popular Python libraries (e.g., Django, scikit-learn, flask).

- **Composition:** The dataset contains 300 verified "Fail-to-Pass" test cases paired with the corresponding code fixes.
- **Task:** The agent is tasked with taking an issue description and the codebase as input, and generating a reproduction script (test) that fails on the buggy version and passes on the fixed version.

- **Gold Standard:** Success is measured by running the generated test against the ground truth "Gold Patch" provided by the dataset maintainers.

### 5.2.2 Baselines

To isolate the impact of the agentic architecture, the proposed solution will be compared against two distinct baselines:

1. **Baseline A (Zero-Shot GPT-4o):** A standard, stateless prompt asking the model to "Write a reproduction script for this issue" without access to the repository structure or execution tools. This represents the current state of "Chat with Code" interfaces.
2. **Baseline B (RAG-Only):** An agent augmented with Retrieval-Augmented Generation (RAG) to access file contents but lacking the "Executor" role. This baseline verifies whether context alone is sufficient, or if execution feedback is truly necessary.

## 5.3 Evaluation Metrics

The performance of the system will be quantified using the following metrics:

- **Pass@1 (%):** The percentage of issues for which the agent generates a valid, passing reproduction script on the first attempt (or after the allowed self-correction turns).
- **Code Coverage (%):** Measured using `coverage.py`, this metric calculates the percentage of lines in the modified files that are executed by the generated test suite.
- **Human Effort (Edits):** A proxy metric estimated by counting the Levenshtein distance between the agent's final generated code and a working solution. Lower distance implies less manual intervention required by the developer.

## 5.4 Execution Procedure

The experiment will follow a strict pipeline executed within a Dockerized sandbox to ensuring reproducibility and safety:

1. **Ingestion:** The "Planner" agent reads the repository structure and the issue description.
2. **Generation:** The "Coder" agent drafts an initial test file.
3. **Validation Loop (The Intervention):**
    - The "Executor" runs the test with `pytest`.
    - If the test fails or errors (e.g., `ImportError`), the stdout/stderr is fed back to the Coder.
    - The Coder attempts to fix the error (Max 5 iterations).
4. **Adjudication:** The final test case is executed against the Gold Patch. If it passes, the instance is marked as "Resolved."

## 5.5 Threats to Validity

### 5.5.1 Internal Validity

The non-deterministic nature of LLMs means results may vary between runs. To mitigate this, experiments will be repeated three times (Temperature = 0.7), and the average Pass@1 reported.

### 5.5.2 External Validity

While SWE-bench represents real-world Python code, the results may not generalize to other languages (e.g., Java, C++) or proprietary logic which differs from open-source patterns. Future work will address this by extending the benchmark to multi-language repositories.

# Chapter 6

# 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.

## 6.1 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.

## 6.2 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.

## 6.3 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.

## 6.4 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.

# References


Ahammad, A., M. El Bajta, and M. Radgui (2025). "MAGISTER: LLM-Based Test Generation with Role-Specialized Agents". In: *2025 International Conference on Intelligent Systems: Theories and Applications (SITA)*.

Bose, D. B., Y. B. Alebachew, and C. Brown (2025). "LLMs in Debate: Does Arguing Make Them Better at Detecting Metamorphic Relations?" In: *2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*.

Bouzenia, I., P. Devanbu, and M. Pradel (2025). "RepairAgent: An Autonomous, LLM-Based Agent for Program Repair". In: *ICSE '25*.

Ding, Y. et al. (2025). "Multi-Agent Auditing for Smart Contracts*". In: *2025 9th International Symposium on Computer Science and Intelligent Control (ISCSIC)*.

Garlapati, A. et al. (2024). "AI-Powered Multi-Agent Framework for Automated Unit Test Case Generation: Enhancing Software Quality through LLM's". In: *2024 5th IEEE Global Conference for Advancement in Technology (GCAT)*.

Hardgrove, I. and J. D. Hastings (2025). "LibLMFuzz: LLM-Augmented Fuzz Target Generation for Black-Box Libraries". In: *2025 Cyber Awareness and Research Symposium (CARS)*.

Hoffmann, J. and D. Frister (2024). "Generating Software Tests for Mobile Applications Using Fine-Tuned Large Language Models". In: *Proc. AST*.

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

Huang, J. (2025). "Research on Multi-Model Fusion Machine Learning Demand Intelligent Forecasting System in Cloud Computing Environment". In: *2025 2nd International Conference on Intelligent Algorithms for Computational Intelligence Systems (IACIS)*.

Jiang, J. et al. (2024). "A Survey on Large Language Models for Code Generation". In: *TOSEM*.

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

Kanagaraj, K. et al. (2025). "LLM-Driven Smart Test Case Generation for Scalable Software Testing". In: *2025 2nd International Conference on Software, Systems and Information Technology (SSITCON)*.

Karanjai, R., L. Xu, and W. Shi (2025). "HPCAgentTester: a Multi-Agent LLM Approach for Enhanced HPC Unit Test Generation". In: *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*.

Kim, Myeongsoo et al. (2025). "A Multi-Agent Approach for REST API Testing with Semantic Graphs and LLM-Driven Inputs". In: *International Conference on Software Engineering (ICSE)*.

Liu, Z. et al. (2025). "Seeing is Believing: Vision-Driven Non-Crash Functional Bug Detection for Mobile Apps". In: *IEEE Transactions on Software Engineering*.

Maklad, Y. et al. (2025). "MultiFuzz: A Dense Retrieval-based Multi-Agent System for Network Protocol Fuzzing". In: *2025 IEEE/ACS 22nd International Conference on Computer Systems and Applications (AICCSA)*.

Nayyeri, A. et al. (2025). "A Privacy-Preserving Recommender for Filling Web Forms Using a Local Large Language Model". In: *Proc. ICCKE*.

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

Pasuksmit, J. et al. (2025). "Human-In-The-Loop Software Development Agents: Challenges and Future Directions". In: *2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR)*.

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

Sallou, J. et al. (2024). "Breaking the Silence: the Threats of Using LLMs in Software Engineering". In: *Proc. ICSE-NIER*.

Salman, I. et al. (2025). "A Vision for Debiasing Confirmation Bias in Software Testing via LLM". In: *2025 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*.

Srinivasan, T. et al. (2025). "DURA-CPS: A Multi-Role Orchestrator for Dependability Assurance in LLM-Enabled Cyber-Physical Systems". In: *Proc. DSN-W*.

Sternak, T. et al. (2025). "Automating Prompt Leakage Attacks on Large Language Models Using Agentic Approach". In: *Proc. MIPRO*.

Stojanović, D. et al. (2024). "Unit Test Generation Multi-Agent AI System for Enhancing Software Documentation and Code Coverage". In: *2024 32nd Telecommunications Forum (TELFOR)*.

Sulaiman, N. A. A. et al. (2025). "An Agentic Reasoning-Based Feedback System for Programming Assignments". In: *2025 IEEE 11th International Conference on Computing, Engineering and Design (ICCED)*.

Tiwari, A. S. et al. (2025). "IntelliTest: An Intelligent Framework for Agentic Functional Test Generation Using Multimodal Data and Domain Knowledge". In: *2025 IEEE Future Networks World Forum (FNWF)*.

Tomic, S., E. Alégroth, and M. Isaac (2025). "Evaluation of the Choice of LLM in a Multi-Agent Solution for GUI-Test Generation". In: *2025 IEEE Conference on Software Testing, Verification and Validation (ICST)*.

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

Wang, Junjie et al. (2023). "Software Testing with Large Language Model: Survey, Landscape, and Vision". In.

Wang, K. et al. (2024). "Do Agents Behave Aligned with Human Instructions? - An Automated Assessment Approach". In: *Proc. QRS-C*.

Wang, S. et al. (2025). "Large Language Model Supply Chain: A Research Agenda". In: *TOSEM '25*.

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

Xia, Chunqiu Steven et al. (2025). "Agentless: Demystifying LLM-based Software Engineering Agents". In: *International Symposium on the Foundations of Software Engineering (FSE)*.

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

Zhao, J. et al. (2024). "Models Are Codes: Towards Measuring Malicious Code Poisoning Attacks on Pre-trained Model Hubs". In: *Proc. ASE*.