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

A Multi-Agent Approach with Golden Examples and Black-Box Observation

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:
[To be defined]

Members:
[To be defined upon thesis defense scheduling]

Porto, February 14, 2026

Statement Of Integrity

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. Artificial Intelligence tools (specifically Large Language Models) were used as assistive tools for literature search, text refinement, and code development assistance. All AI-generated content was reviewed, validated, and integrated by the author with full responsibility for the final output.

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

ISEP, Porto, February 14, 2026

Dedicatory

To my family, for their unwavering support and patience throughout this journey.

To all software engineers striving to build more reliable and secure systems.

Abstract

The manual creation of software tests is costly, slow, and difficult to scale when seeking comprehensive coverage of complex applications. This dissertation addresses the problem through automated test generation supported by Multi-Agent Systems (MAS) powered by Large Language Models (LLMs), combining two complementary approaches: learning from existing test examples and observing application behaviour through black-box traffic capture.

The central contribution is the implementation of a modular platform, TestForge, that integrates six specialised agents (Observer, Mapper, Analyser, Generator, Executor, Validator) in a dual-pipeline architecture. The Golden Examples pipeline analyses existing test files using Tree-sitter AST parsing to extract patterns, conventions, and testing strategies, then uses LLM few-shot prompting to generate new tests that go beyond trivial assertions—targeting state integrity, boundary conditions, business logic, and error handling. The Black-Box Observer pipeline captures HTTP traffic from running applications, maps the API surface through endpoint normalisation and schema inference, and generates tests without requiring source code access.

A persistent memory layer, implemented through the Letta framework with local LLM inference (Ollama), enables the system to progressively learn about target applications across sessions. The platform runs entirely locally, ensuring that no application code or test data leaves the user's environment.

Following the PRISMA methodology, a systematic literature review of 55 studies covers MAS architectures for testing, LLM-driven test generation, and security considerations. Experimental evaluation on a Flask CRUD API demonstrates that the Golden Examples pipeline generates insightful tests: of 9 generated tests, 4 found genuine application deficiencies (validation ordering bugs, missing type checks, inconsistent error handling) that were not caught by the 23 original golden tests. The results highlight the potential of combining AST-based pattern extraction with LLM generation for producing high-quality automated tests.

Keywords: Multi-Agent Systems, Large Language Models, Automated Test Generation, Golden Examples, Black-Box Testing, Few-Shot Learning

Resumo

A criação manual de testes de software é dispendiosa, lenta e difícil de escalar quando se pretende uma cobertura abrangente de aplicações complexas. Esta dissertação aborda o problema através da geração automatizada de testes suportada por Sistemas Multi-Agente (MAS) alimentados por Modelos de Linguagem de Grande Escala (LLMs), combinando duas abordagens complementares: aprendizagem a partir de exemplos de testes existentes e observação do comportamento da aplicação através de captura de tráfego black-box.

O contributo central é a implementação de uma plataforma modular, TestForge, que integra seis agentes especializados (Observador, Mapeador, Analisador, Gerador, Executor, Validador) numa arquitetura de duplo pipeline. O pipeline de Exemplos Golden analisa ficheiros de teste existentes usando análise AST via Tree-sitter para extrair padrões, convenções e estratégias de teste, utilizando depois prompting few-shot com LLMs para gerar novos testes que vão além de asserções triviais—visando integridade de estado, condições limite, lógica de negócio e tratamento de erros. O pipeline de Observação Black-Box captura tráfego HTTP de aplicações em execução, mapeia a superfície API através de normalização de endpoints e inferência de esquemas, e gera testes sem necessitar de acesso ao código-fonte.

Uma camada de memória persistente, implementada através do framework Letta com inferência LLM local (Ollama), permite ao sistema aprender progressivamente sobre as aplicações alvo entre sessões. A plataforma executa inteiramente de forma local, garantindo que nenhum código ou dados de teste saem do ambiente do utilizador.

Seguindo a metodologia PRISMA, uma revisão sistemática da literatura de 55 estudos cobre arquiteturas MAS para testes, geração de testes baseada em LLMs e considerações de segurança. A avaliação experimental numa API Flask CRUD demonstra que o pipeline de Exemplos Golden gera testes perspicazes: de 9 testes gerados, 4 encontraram deficiências genuínas na aplicação (erros de ordenação de validação, verificações de tipo em falta, tratamento inconsistente de erros) que não foram detetadas pelos 23 testes golden originais. Os resultados evidenciam o potencial da combinação de extração de padrões baseada em AST com geração LLM para a produção de testes automatizados de alta qualidade.

Palavras-chave: Multi-Agent Systems, Large Language Models, Automated Test Generation, Golden Examples, Black-Box Testing, Few-Shot Learning

Acknowledgement

I would like to express my sincere gratitude to my supervisor, Professor Constantino Martins, for his guidance, expertise, and continuous support throughout this research. His insights into artificial intelligence and software engineering have been invaluable in shaping this work.

I am grateful to the faculty and staff of the Department of Computer Engineering at ISEP for providing an excellent academic environment and the resources necessary to conduct this research.

I would also like to thank my colleagues in the MEIA program for the stimulating discussions and collaborative spirit that enriched my learning experience.

Finally, I extend my deepest appreciation to my family and friends for their understanding, encouragement, and support during the demanding periods of this thesis work.

Contents

List of Source Code **xxi**

List of Acronyms **xxv**

1 Introduction **1**
1.1 Contextualisation . . . 1
1.2 Description of the Problem . . . 2
1.3 Research Questions and Objectives . . . 3
1.3.1 Thesis Objective . . . 3
1.3.2 Milestones . . . 3
1.4 Contributions of the Dissertation . . . 4
1.5 Document Structure . . . 4

2 Systematic Review of the Literature **7**
2.1 Methodology . . . 7
2.1.1 Review Question and Research Questions . . . 7
2.1.2 Search Strategy . . . 8
2.1.3 Inclusion and Exclusion Criteria . . . 9
2.1.4 Study Selection Process . . . 9
2.1.5 Quality Assessment . . . 10
2.1.6 Data Extraction . . . 11
2.2 Results . . . 11
2.2.1 Study Selection and Characteristics . . . 12
2.2.2 RQ1, Golden Examples and Few-Shot Test Generation . . . 15
2.2.3 RQ2, Black-Box Testing and Traffic Analysis . . . 17
2.2.4 RQ3, Multi-Agent Systems for Software Testing . . . 17
2.2.5 RQ4, Persistent Agent Memory . . . 20
2.2.6 RQ5, Practical Deployment Considerations . . . 21
2.3 Critical Discussion . . . 25
2.3.1 RQ1, On the Limits of Few-Shot Test Generation . . . 25
2.3.2 RQ2, On the Untapped Potential of Traffic Observation . . . 26
2.3.3 RQ3, On Agent Architecture Trade-Offs . . . 26
2.3.4 RQ4, On the Absence of Memory in Testing Systems . . . 27
2.3.5 RQ5, On the Privacy-Capability Trade-Off . . . 27
2.3.6 Research Gaps . . . 27
2.3.7 Implications for the Proposed Research . . . 28
2.4 Conclusions of the Review . . . 28
2.4.1 Limitations of This Review . . . 29
2.4.2 Future Directions . . . 29

3 Data Collection and Pre-processing **31**

3.1 Data Sources and Selection . . . 31
3.1.1 Selection Criteria and Objectives . . . 31
3.1.2 Golden Test Examples . . . 31
3.1.3 HTTP Traffic Captures . . . 32
3.2 Pre-processing: Golden Examples Pipeline . . . 32
3.2.1 AST Parsing with Tree-sitter . . . 32
3.2.2 HTTP Endpoint Detection . . . 32
3.2.3 Pattern Aggregation into Style Guide . . . 33
3.3 Pre-processing: Observer Pipeline . . . 33
3.3.1 HTTP Exchange Normalisation . . . 33
3.3.2 Endpoint Mapping and Schema Inference . . . 33
3.4 Data Representation Models . . . 34
3.5 Risk Analysis and Data Bias . . . 35
3.5.1 Golden Example Bias . . . 35
3.5.2 Observer Data Limitations . . . 35
3.5.3 Mitigation Strategies . . . 35

4 Methods and Tools **37**
4.1 System Overview and Architecture . . . 37
4.2 Multi-Agent Architecture . . . 37
4.2.1 Agent Roles and Responsibilities . . . 38
4.2.2 Agent Communication . . . 38
4.2.3 Base Agent Design . . . 38
4.3 Golden Examples Pipeline Design . . . 38
4.3.1 Few-Shot Prompting Strategy . . . 39
4.4 Black-Box Observer Pipeline Design . . . 39
4.5 Persistent Memory via Letta . . . 40
4.5.1 Memory Architecture . . . 40
4.5.2 Custom Tools . . . 40
4.5.3 Local Execution . . . 40
4.6 Orchestration Engine . . . 41
4.7 Development Stack and Deployment Environment . . . 41
4.7.1 LLM Configuration . . . 41

5 Solution Implementation **43**
5.1 Development Environment and Repository Structure . . . 43
5.2 Implementation of the Analyser Agent . . . 44
5.2.1 Tree-sitter Integration . . . 44
5.2.2 HTTP Method and Endpoint Extraction . . . 44
5.2.3 Style Guide Aggregation . . . 45
5.3 Implementation of the Generator Agent . . . 45
5.3.1 Prompt Construction . . . 45
5.3.2 Response Parsing . . . 45
5.4 Implementation of the Observer and Mapper Agents . . . 46
5.4.1 Observer Agent . . . 46
5.4.2 Mapper Agent . . . 46
5.5 Implementation of the Executor Agent . . . 47
5.5.1 Auto-generated conftest.py . . . 47
5.5.2 Pytest Output Parsing . . . 47

5.6 Implementation of the Validator Agent . . . 47
5.7 Streamlit Web Interface . . . 48
5.8 Implementation Challenges and Solutions . . . 49

6 Experimentation and Discussion **51**
6.1 Experimental Setup and Methodology . . . 51
6.1.1 Target Application . . . 51
6.1.2 Golden Test Examples . . . 51
6.1.3 Experimental Environment . . . 52
6.2 Evaluation Metrics . . . 52
6.2.1 Test Quality Metrics . . . 52
6.2.2 Quality Scoring Dimensions . . . 53
6.2.3 Structural Metrics . . . 54
6.3 Results and Analysis . . . 55
6.3.1 Golden Examples Pipeline Results . . . 55
Analyser Output . . . 55
Generated Tests . . . 55
Execution Results . . . 56
Analysis of Failures . . . 56
6.3.2 Quality Assessment . . . 56
6.4 Discussion and Analysis of Results . . . 56
6.4.1 Effectiveness of the Golden Examples Approach . . . 57
6.4.2 Test Quality vs. Pass Rate . . . 57
6.4.3 Observations on the Black-Box Observer Pipeline . . . 57
6.4.4 Observations on the Multi-Agent Architecture . . . 58
6.4.5 Observations on Persistent Agent Memory . . . 58
6.4.6 Observations on Local LLM Execution . . . 58
6.4.7 Limitations of the Current Evaluation . . . 59
6.4.8 Summary of Findings . . . 59

7 Data Protection, Security and Ethics **61**
7.1 Current State of the Literature . . . 61
7.2 Privacy-by-Design in Test Generation . . . 61
7.2.1 Local Execution as Primary Mitigation . . . 61
7.2.2 Data Minimisation . . . 62
7.2.3 Pseudonymisation and Identity Management . . . 62
7.3 Security and System Integrity . . . 62
7.3.1 Execution Isolation . . . 62
7.3.2 Prompt Injection Resilience . . . 62
7.4 Ethical Considerations . . . 62
7.4.1 Transparency . . . 62
7.4.2 Human Oversight . . . 63
7.4.3 AI Tool Disclosure . . . 63
7.5 Regulatory Compliance . . . 63
7.5.1 General Data Protection Regulation (GDPR) . . . 63
7.5.2 EU AI Act . . . 63

8 Conclusions and Future Work **65**
8.1 Conclusions . . . 65

8.1.1 Summary of Contributions 66
8.2 Limitations 66
8.3 Next Iteration and Future Work 67
8.3.1 Planned for the Next Iteration 67
8.3.2 Longer-Term Research Directions 67

Bibliography **69**

A Extended Code Listings **73**
A.1 Configuration Schema 73
A.2 LLM Gateway Implementation 74
A.3 Prompt Templates 76
A.4 Security Components (Design Specification) 78
A.5 Docker Sandbox Configuration (Design Specification) 80
A.6 CLI Implementation (Design Specification) 82

B Detailed Experimental Results **85**
B.1 Target Application Endpoints 85
B.2 Golden Test Examples Summary 85
B.3 Analyser Agent Output 86
B.4 Generated Test Details 86
B.5 Validator Agent Scores 86
B.6 Bug Classification 87
B.7 Endpoint Coverage Analysis 87
B.8 Insightful Category Coverage 87

List of Figures

2.1 PRISMA 2020 flow diagram of the study selection process 12

4.1 High-level architecture of TestForge showing the dual-pipeline design. . . . 37
4.2 Data flow through the Golden Examples pipeline. Blue-tinted nodes represent stages specific to this pipeline; orange, red, and purple nodes represent shared stages. 39
4.3 Data flow through the Black-Box Observer pipeline. Green-tinted nodes represent stages specific to this pipeline; orange, red, and purple nodes represent shared stages. 39

List of Tables

2.1 Inclusion criteria 10
2.2 Exclusion criteria 10
2.3 Quality assessment criteria 11
2.4 Data extraction form template 11
2.5 PRISMA study selection summary 13
2.6 Detailed search results by database and query 14
2.7 Full-text exclusion reasons 14
2.8 Quality scores for selected studies 15
2.9 Distribution of studies by publication year 15
2.10 Distribution of studies by venue type 16
2.11 Number of studies addressing each research question 16
2.12 Comparison of multi-agent frameworks for software engineering 19
2.13 Detailed MAS framework comparison 19
2.14 Comprehensive security threat catalog 24

4.1 Technology stack of the TestForge platform. 41

6.1 Experimental environment configuration. 52
6.2 Generated tests from the Golden Examples pipeline. 55
6.3 Quality metrics for the Golden Examples pipeline run. 56

B.1 Target application endpoints and their expected behaviour 85
B.2 Golden test files: per-file breakdown 85
B.3 Extracted style guide details 86
B.4 Per-test details of the 9 generated tests 86
B.5 Validator Agent quality scores per test 87
B.6 Classification of test failures 87
B.7 Endpoint coverage: golden tests vs. generated tests 88
B.8 Test category coverage 88

List of Source Code

5.1 Repository structure of the TestForge platform. 43
5.2 Tree-sitter AST parsing of test files (simplified). 44
5.3 Endpoint extraction from f-string URLs. 44
5.4 Path normalisation in the Mapper Agent. 46
A.1 System configuration schema (config/schema.py) 73
A.2 LLM Gateway with provider abstraction (llm/gateway.py) 74
A.3 Prompt templates for test generation (llm/prompts/test_generation.py) . 76
A.4 PII Scrubber design specification (security/scrubber.py) 78
A.5 Dockerfile for secure sandbox environment (docker/Dockerfile.sandbox) . . 80
A.6 Seccomp security profile (docker/seccomp-profile.json) 81
A.7 Command-line interface design specification (integration/cli.py) 82

List of Symbols

LC	Line Coverage	%
BC	Branch Coverage	%
MS	Mutation Score	%
CR	Compilation Rate	%
ER	Execution Rate	%
PR	Pass Rate	%
AD	Assertion Density	assertions/test
k	Number of generation attempts (pass@k)	–
n	Total number of samples	–
c	Number of correct samples	–

List of Acronyms

ACI	Agent-Computer Interface.
AST	Abstract Syntax Tree.
CoT	Chain-of-Thought.
GDPR	General Data Protection Regulation.
LLM	Large Language Model.
MAS	Multi-Agent System.
PII	Personally Identifiable Information.
PRISMA	Preferred Reporting Items for Systematic Reviews and Meta-Analyses.
RAG	Retrieval-Augmented Generation.
SDLC	Software Development Life Cycle.
SUT	System Under Test.

Chapter 1

Introduction

1.1 Contextualisation

This section situates the dissertation within the broader landscape of software testing, tracing the evolution from manual testing practices to LLM-powered multi-agent approaches and motivating the need for the research presented in subsequent chapters.

Software testing remains one of the most critical yet resource-intensive activities in the Software Development Life Cycle (SDLC). Industry estimates consistently indicate that testing activities consume between 30% and 50% of total development effort (Ammann and Offutt 2016), while inadequate testing continues to cost the global economy billions annually through software failures, security breaches, and system downtime. According to the Consortium for Information and Software Quality (CISQ), the cost of poor software quality in the United States alone exceeded $2.4 trillion in 2022, with a substantial portion attributable to inadequate testing practices (Krasner 2022). Organizations allocate an average of 23% of their IT budgets to quality assurance and testing activities, yet many report dissatisfaction with their testing coverage and effectiveness.

The rise of agile and DevOps practices has intensified these challenges. Modern development methodologies emphasize rapid iteration, continuous integration, and frequent releases, often multiple times per day. This acceleration places enormous pressure on testing processes, which must keep pace with development velocity while maintaining quality standards. Traditional manual testing approaches simply cannot scale to meet these demands, and even script-based automation requires substantial upfront investment and ongoing maintenance as the System Under Test (SUT) evolves.

The emergence of Large Language Model (LLM) technology has opened fundamentally new possibilities for addressing these challenges. Models such as GPT-4, Claude, and open-weight alternatives like LLaMA (Brown et al. 2020) have demonstrated remarkable capabilities in understanding and generating code, comprehending natural language specifications, and reasoning about software behaviour (Fan et al. 2023). Unlike prior automated test generation techniques, such as search-based approaches (EvoSuite (Fraser and Arcuri 2011), Randoop), which operate on syntactic or structural properties and suffer from the "oracle problem", LLM-based approaches can understand code semantics, infer intended behaviour from context, and generate human-readable test code with meaningful assertions.

However, the complexity of comprehensive software testing often exceeds the capabilities of single-model approaches. Real-world testing involves multiple interrelated activities: understanding the target application, identifying testable surfaces, generating appropriate test cases, executing tests, and validating results. This complexity motivates the exploration of

Multi-Agent System (MAS) architectures, where multiple specialised agents collaborate to address different aspects of the testing challenge. Recent frameworks such as MetaGPT (Hong et al. 2023), ChatDev (Qian et al. 2024), and SWE-agent (Yang et al. 2024) have demonstrated the viability of multi-agent approaches for software engineering tasks, employing hierarchical or peer-to-peer coordination models with role-based specialisation.

This dissertation investigates how a multi-agent system, combining LLM-powered test generation with two complementary input strategies, learning from existing test examples and observing application behaviour, can automate the production of high-quality integration and end-to-end tests. The central contribution is **TestForge**, a modular platform that implements both approaches and integrates persistent agent memory for progressive learning across sessions.

1.2 Description of the Problem

This section identifies the key challenges that current automated test generation approaches face, from shallow test quality and lack of application context to input modality constraints and privacy concerns, establishing the problem space that this dissertation addresses.

Despite significant advances in LLM capabilities, automated test generation faces several fundamental challenges that limit practical adoption. The most immediate concern is the quality of generated tests: most existing tools produce shallow "smoke tests" that verify basic connectivity, for example, confirming that an endpoint returns a 200 status code, without probing state integrity, boundary conditions, business logic violations, or data consistency. The resulting tests find few real bugs and provide a false sense of coverage.

A closely related issue is the lack of application context. Single-shot generation approaches treat each invocation independently, with no memory of previous interactions, discovered endpoints, or known patterns. This prevents the system from building a progressively deeper understanding of the target application over time. The problem is compounded by what can be termed the cold-start challenge: when no reference tests exist, generators lack information about project conventions, assertion styles, fixture patterns, and the testing framework in use. Conversely, when high-quality reference tests are available, current tools fail to leverage them effectively as few-shot examples.

Beyond test quality, existing approaches suffer from input modality limitations. Most tools require explicit endpoint definitions, OpenAPI specifications, or direct source code access. Many real-world applications, however, lack formal API documentation, making it difficult to determine what needs testing without observing the application in operation. Furthermore, comprehensive test generation requires multiple distinct activities (analysis, generation, execution, and validation) that benefit from specialised treatment, yet monolithic single-agent systems struggle to coordinate this complexity effectively. Finally, organisations handling sensitive data face legitimate concerns about transmitting proprietary code to cloud-based LLM providers, and the evolving regulatory landscape surrounding the GDPR and the EU AI Act adds further constraints that motivate local execution capabilities.

This dissertation addresses these challenges through a dual-pipeline approach implemented as a multi-agent platform. The **Golden Examples Pipeline** analyses existing high-quality test files using Abstract Syntax Tree (AST) parsing to extract patterns, conventions, and testing strategies, then uses few-shot prompting to generate new tests that follow the same style while targeting untested scenarios. Complementing this, the **Black-Box Observer**

Pipeline captures HTTP traffic from a running application (via proxy interception or HAR file import), maps discovered endpoints and their schemas, and generates tests based on observed behaviour without requiring source code access. Both pipelines are orchestrated by a multi-agent system where specialised agents handle analysis, generation, execution, and validation. A persistent memory layer, implemented through the Letta framework, enables the system to accumulate knowledge about target applications across sessions, progressively improving test quality.

1.3 Research Questions and Objectives

This section presents the central research question that drives the thesis, the objective that the dissertation aims to fulfil, and the milestones that structure the research programme.

The thesis is structured around an overarching investigation question that guides the systematic review and the entire research programme:

> *How can a multi-agent system, powered by large language models and informed by both existing test examples and observed application behaviour, automate the generation of high-quality integration and end-to-end tests while preserving data privacy?*

This question will be answered progressively throughout the dissertation: Chapter 2 surveys the state of the art and decomposes the question into specific review themes, Chapter 3 and Chapter 4 describe the data strategies and architectural design adopted, Chapter 5 presents the prototype implementation, and Chapter 6 evaluates the results empirically.

1.3.1 Thesis Objective

The objective of this dissertation is to design, implement, and evaluate a multi-agent platform, TestForge, that automates the generation of high-quality integration and end-to-end tests for web applications. The platform must support two complementary input strategies: a golden examples pipeline that learns testing patterns from existing high-quality test files through Abstract Syntax Tree (AST) analysis and few-shot prompting, and a black-box observer pipeline that discovers API endpoints by capturing HTTP traffic from a running application and generates tests from observed behaviour without requiring source code access. The system must coordinate multiple specialised agents across analysis, generation, execution, and validation stages, while incorporating persistent memory so that the platform accumulates knowledge about target applications across sessions. Critically, the entire platform, including LLM inference, must be capable of running on local infrastructure using open-weight models, ensuring that organisations retain full control over their proprietary code and comply with data protection regulations such as the GDPR and the EU AI Act. The thesis validates this objective through a systematic literature review following Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) methodology and an empirical evaluation against a real-world application, measuring test quality, bug detection capability, and practical deployment feasibility.

1.3.2 Milestones

The research programme is organised around six milestones, each corresponding to a verifiable deliverable:

1. **M1: Systematic Literature Review**: Completion of the PRISMA-based review covering MAS architectures, LLM-driven testing, and security considerations, resulting in a synthesis of 55 studies and identification of research gaps.
2. **M2: Platform Architecture**: Definition of the dual-pipeline architecture with six specialised agents, typed inter-agent communication, and persistent memory integration.
3. **M3: Golden Examples Pipeline**: Implementation and validation of AST-based pattern extraction (Tree-sitter) and LLM few-shot test generation from reference test files.
4. **M4: Black-Box Observer Pipeline**: Implementation and validation of HTTP traffic capture, endpoint mapping, and LLM-driven test generation from observed behaviour.
5. **M5: Experimental Evaluation**: End-to-end evaluation of the platform against a real-world application, measuring test quality, bug detection, and generation effectiveness.
6. **M6: Practical Deployment Assessment**: Evaluation of local execution feasibility, privacy compliance, and cost-effectiveness compared to cloud-based alternatives.

1.4 Contributions of the Dissertation

This section summarises the main contributions that emerge from the research programme.

The first contribution is a systematic literature review of 55 studies, conducted following PRISMA methodology, covering MAS architectures for testing, LLM-driven test generation, security considerations, and practical deployment challenges. This review identifies research gaps and establishes the theoretical foundation for the platform design.

The second contribution is the golden examples approach, a method for extracting testing patterns from existing test suites using Tree-sitter AST parsing and leveraging them as few-shot examples for LLM-based generation. This approach produces insightful tests that target state integrity, boundary conditions, and business logic rather than shallow smoke tests.

The third contribution is the black-box observer approach, which automatically maps API endpoints from captured HTTP traffic and generates test suites without requiring source code access or formal API specifications.

The fourth and central contribution is TestForge itself: a modular, open-source multi-agent platform implementing both approaches with six specialised agents (Observer, Mapper, Analyser, Generator, Executor, Validator), persistent memory via Letta, and support for fully local execution using open-weight LLMs. The fifth contribution is an empirical evaluation demonstrating the platform's effectiveness on a real-world application, with analysis of test quality, bug detection capabilities, and the impact of progressive learning. Finally, the sixth contribution provides a practical assessment of privacy, performance, and cost trade-offs for deploying LLM-based test generation on local infrastructure, including considerations related to the GDPR and the EU AI Act regulatory framework.

1.5 Document Structure

This section provides a roadmap of the dissertation, outlining the purpose and content of each chapter so the reader can navigate the document effectively.

This thesis is organised into eight chapters that progressively address the research questions and objectives. Following this introduction, Chapter 2 presents a systematic review of the literature following PRISMA methodology, covering MAS architectures for testing, LLM-driven test generation, example-based learning approaches, black-box testing techniques, and security considerations. Chapter 3 describes the data sources used by both pipelines, including golden test examples with AST-based pattern extraction, HTTP traffic capture and HAR file processing, and the data representations applied before generation. Chapter 4 presents the system architecture, including the dual-pipeline design, the six-agent multi-agent architecture, LLM integration strategy, and persistent memory via Letta. Chapter 5 details the prototype implementation of TestForge, covering the development environment, implementation of each agent, the Streamlit web interface, and challenges encountered. Chapter 6 presents the experimental evaluation, including the setup, metrics, results from the Golden Examples pipeline, and discussion of findings in relation to the research questions. Chapter 7 discusses privacy-by-design considerations, security measures, GDPR compliance, and ethical implications. Finally, Chapter 8 summarises the contributions, answers the research questions, discusses limitations, and outlines directions for future research. Appendices provide supplementary materials including PRISMA data extraction tables, extended code listings, and additional experimental results.

Chapter 2

Systematic Review of the Literature

This chapter presents a comprehensive systematic literature review following the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) methodology to ensure rigour, transparency, and reproducibility. The review is guided by its own overarching research question, distinct from the thesis question presented in Chapter 1, which surveys how artificial intelligence techniques have been applied to automated software test generation. This question is decomposed into five research questions (RQ1–RQ5) that structure the search strategy, thematic analysis, and synthesis. The chapter begins with a detailed description of the review methodology (Section 2.1), presents the results organised by research question with the full thematic analysis for each (Section 2.2), provides a critical discussion of the findings and identification of research gaps (Section 2.3), and concludes with a summary of the review, its limitations, and directions for future work (Section 2.4).

2.1 Methodology

This systematic literature review follows the PRISMA 2020 guidelines (Page et al. 2021) and incorporates recommendations from established software engineering research methodology (Kitchenham and Charters 2007). The methodology encompasses research question formulation, search strategy definition, study selection criteria, quality assessment, and data extraction procedures.

2.1.1 Review Question and Research Questions

The systematic review is guided by its own overarching research question, distinct from the thesis question presented in Chapter 1. While the thesis question concerns the design and evaluation of a specific platform (TestForge), the review question surveys the broader state of the art in AI-driven test generation to establish the theoretical foundation upon which the thesis builds:

> *How have artificial intelligence techniques, particularly large language models and multi-agent systems, been applied to automated software test generation, and what are the current capabilities, limitations, and open challenges?*

To answer this question systematically, the review decomposes it into five research questions (RQ). Each question targets a distinct aspect of the problem space, and together they provide a comprehensive mapping of the existing literature that informs the design decisions made in subsequent chapters.

RQ1 asks: *What techniques have been used to leverage existing test examples and few-shot prompting for AI-driven test generation, and what quality improvements do they achieve?*

This question examines how reference tests, code demonstrations, and in-context learning strategies have been employed to guide LLMs toward generating higher-quality tests, covering retrieval-based example selection, mutation-guided feedback, and AST-based pattern extraction.

RQ2 asks: *How have black-box testing and traffic analysis methods been combined with AI for automatic endpoint discovery and test suite creation?* This question surveys techniques for testing APIs without source code access, including stateful fuzzing, specification inference, and the emerging use of LLMs for generating tests from observed HTTP exchanges.

RQ3 asks: *What multi-agent system architectures have been proposed for software engineering and testing tasks, and how do coordination models, role specialisation, and communication patterns affect effectiveness?* This question investigates the design space for multi-agent frameworks, comparing hierarchical, peer-to-peer, and hybrid coordination models, and identifying the role decomposition patterns most relevant to automated testing.

RQ4 asks: *How have persistent memory mechanisms been used in LLM-based agent systems, and what is their potential for improving test generation quality across sessions?* This question explores memory architectures for language agents, including virtual context management, reflective memory, and retrieval-augmented generation, and evaluates whether they have been applied to software testing.

RQ5 asks: *What are the practical considerations, including privacy, security, cost, and regulatory compliance, for deploying AI-based test generation systems, particularly with local execution?* This question addresses the deployment landscape, covering the trade-offs between proprietary and open-weight models, security threats specific to agent-based systems, mitigation strategies, and the requirements imposed by the GDPR and the EU AI Act.

The search strategy, inclusion criteria, and thematic analysis sections that follow are organised around these five research questions. The Results section consolidates the findings and thematic analysis for each question, the Critical Discussion provides a critical analysis per question, and the Conclusions section summarises the state of the field and identifies the research gaps that motivate this thesis.

2.1.2 Search Strategy

The literature search employed multiple complementary search strings targeting different aspects of the research questions. Searches were conducted across IEEE Xplore, ACM Digital Library, Scopus, and arXiv to capture both peer-reviewed publications and recent preprints in this rapidly evolving field.

The first search string (S1) targeted multi-agent systems and test generation, addressing RQ1 and RQ3:

```
("Multi-Agent" OR "MAS" OR "LLM" OR "Large Language Model"
 OR "GPT" OR "Code Generation")
AND
("Software Testing" OR "Test Generation" OR "Automated Testing"
 OR "Unit Testing" OR "Integration Testing" OR "End-to-End Testing")
```

The second search string (S2) focused on few-shot and example-driven testing for RQ1:

```
("Few-Shot" OR "In-Context Learning" OR "Example-Driven"
```

```
 OR "Example-Based" OR "Prompt Engineering")
AND
("Test Generation" OR "Test Case" OR "Unit Test" OR "Code Generation")
AND
("LLM" OR "Large Language Model" OR "GPT" OR "Codex" OR "Copilot")
```

The third search string (S3) targeted black-box testing and API analysis for RQ2:

```
("Black-Box" OR "Black Box" OR "REST API" OR "API Testing"
 OR "Traffic Analysis" OR "HTTP" OR "API Discovery")
AND
("Test Generation" OR "Automated Testing" OR "Fuzzing"
 OR "Specification Inference" OR "OpenAPI")
```

The fourth search string (S4) addressed agent memory and stateful systems for RQ4:

```
("Agent Memory" OR "Persistent Memory" OR "Long-Term Memory"
 OR "Stateful Agent" OR "MemGPT" OR "Memory-Augmented")
AND
("LLM" OR "Large Language Model" OR "Agent" OR "Multi-Agent")
```

The fifth search string (S5) covered architecture, effectiveness, and deployment for RQ3 and RQ5:

```
("Multi-Agent" OR "Agent Framework" OR "LLM Agent")
AND
("Architecture" OR "Effectiveness" OR "Performance"
 OR "Benchmark" OR "Evaluation" OR "Security" OR "Privacy"
 OR "Local Deployment" OR "Open-Weight")
```

The search period covered publications from January 2020 to January 2026, capturing the emergence of modern LLMs (beginning with GPT-3) through to current developments. Searches were conducted across IEEE Xplore, ACM Digital Library, Scopus, and arXiv. Reference lists of included studies were manually screened to identify additional relevant works (snowballing).

2.1.3 Inclusion and Exclusion Criteria

To ensure that only relevant and methodologically sound studies were retained, a set of inclusion and exclusion criteria was defined prior to screening. The inclusion criteria were designed to capture the full breadth of the five research questions, encompassing multi-agent architectures, LLM-based test generation, example-driven approaches, black-box testing, agent memory, and practical deployment concerns. The exclusion criteria filtered out studies that fell outside the scope of the review or that lacked the methodological rigour necessary for meaningful synthesis. Both sets of criteria were applied consistently at the title/abstract screening and full-text assessment stages. Table 2.1 and Table 2.2 present the complete criteria.

2.1.4 Study Selection Process

The study selection followed a multi-stage screening process designed to progressively narrow the pool of candidate studies while maintaining transparency and reproducibility. In the first stage, titles and abstracts were reviewed against the inclusion and exclusion criteria

Table 2.1: Inclusion criteria

ID	Criterion
IC1	Studies presenting MAS architectures for software testing or development
IC2	Studies with empirical evaluation of LLM-based testing tools
IC3	Studies reporting quantitative metrics (coverage, bug detection, pass@k)
IC4	Framework papers describing multi-agent systems (MetaGPT, ChatDev, SWE-agent, etc.)
IC5	Studies on few-shot, example-driven, or in-context learning for code/test generation
IC6	Studies on black-box testing, API discovery, or traffic-based test generation
IC7	Studies on agent memory architectures or stateful LLM agents
IC8	Studies addressing security, privacy, or practical deployment concerns in LLM-based systems

Table 2.2: Exclusion criteria

ID	Criterion
EC1	Single-agent LLM approaches without multi-agent coordination (unless providing essential baseline comparisons)
EC2	Studies lacking empirical validation or technical depth
EC3	Non-English publications
EC4	Opinion pieces, editorials, or short papers without substantive technical content
EC5	Studies focused exclusively on non-testing applications (e.g., pure code generation without testing)
EC6	Duplicate publications or extended versions superseded by later work

defined above; this screening was performed independently by two reviewers, with disagreements resolved through discussion. Studies that passed the initial screening proceeded to a full-text assessment, during which each article was read in its entirety and evaluated against the complete set of criteria, with explicit documentation of the reason for any exclusion. Following this, a quality assessment was applied to the remaining studies to evaluate methodological rigour, clarity of reporting, and validity of conclusions. Finally, a snowballing step, encompassing both forward and backward reference searching from the included studies, was conducted to identify additional relevant works that may have been missed by the database searches.

2.1.5 Quality Assessment

Each included study was assessed using an adapted quality checklist comprising seven criteria (Table 2.3): whether the study presents a clear statement of research objectives, whether it employs an appropriate research methodology, whether the experimental setup is described in sufficient detail, whether the evaluation metrics are valid and reliable, whether appropriate statistical analysis is applied where applicable, whether the authors discuss limitations and threats to validity, and whether the results are reproducible. Each criterion was scored as fully met (1), partially met (0.5), or not met (0), yielding a maximum score of 7. Studies scoring below 4 were flagged for careful consideration of their contribution weight in the synthesis, though none were excluded solely on this basis.

Table 2.3: Quality assessment criteria

ID	Criterion	Scoring
QC1	Clear research objectives	0/0.5/1
QC2	Appropriate methodology	0/0.5/1
QC3	Adequate experimental setup description	0/0.5/1
QC4	Valid and reliable metrics	0/0.5/1
QC5	Appropriate statistical analysis	0/0.5/1
QC6	Discussion of limitations	0/0.5/1
QC7	Reproducibility of results	0/0.5/1

2.1.6 Data Extraction

A structured data extraction form was applied to each included study (Table 2.4). For every study, the form captured bibliographic details (authors, year, venue, and publication type), the review themes addressed, and the research methodology employed (empirical study, framework proposal, survey, or case study). Where applicable, the form recorded the agent architecture, including the coordination model, the number and roles of agents, and the communication patterns used. The specific LLMs employed, model names, versions, and configurations, were noted alongside the evaluation metrics and quantitative results reported. Security and privacy concerns identified by the authors were extracted together with any mitigation strategies proposed. Finally, the form recorded the limitations acknowledged by each study and the key findings relevant to the review themes.

Table 2.4: Data extraction form template

Field	Description
Study ID	Unique identifier (S01–S55)
Authors	Author names
Year	Publication year
Venue	Conference/journal name
Type	Conference paper / Journal article / Preprint
RQs Addressed	Which of RQ1–RQ5 the study addresses
Methodology	Empirical study / Framework / Survey / Case study
Agent Architecture	Coordination model, number of agents, roles
LLMs Used	Model names, versions, configurations
Evaluation Metrics	Coverage, pass@k, mutation score, etc.
Benchmarks Used	HumanEval, SWE-bench, Defects4J, etc.
Security Concerns	Identified risks and vulnerabilities
Mitigations Proposed	Countermeasures and defensive techniques
Key Findings	Primary results and conclusions
Limitations	Acknowledged limitations

2.2 Results

This section presents the findings of the systematic review, organised by the five research questions introduced in the methodology. The first subsection describes the study selection process and the characteristics of the 55 included studies. The subsequent subsections

present the full thematic analysis for each research question, synthesising the relevant evidence from the corpus.

2.2.1 Study Selection and Characteristics

The systematic search identified a substantial number of potentially relevant records across the searched databases. After removing duplicates, records underwent title and abstract screening. Studies clearly not meeting inclusion criteria were excluded, with the remaining records proceeding to full-text assessment. Figure 2.1 illustrates the complete study selection process following the PRISMA 2020 flow diagram, and Table 2.5 summarizes the numerical breakdown at each stage. Table 2.6 presents the detailed search results from each database, broken down by search string group.

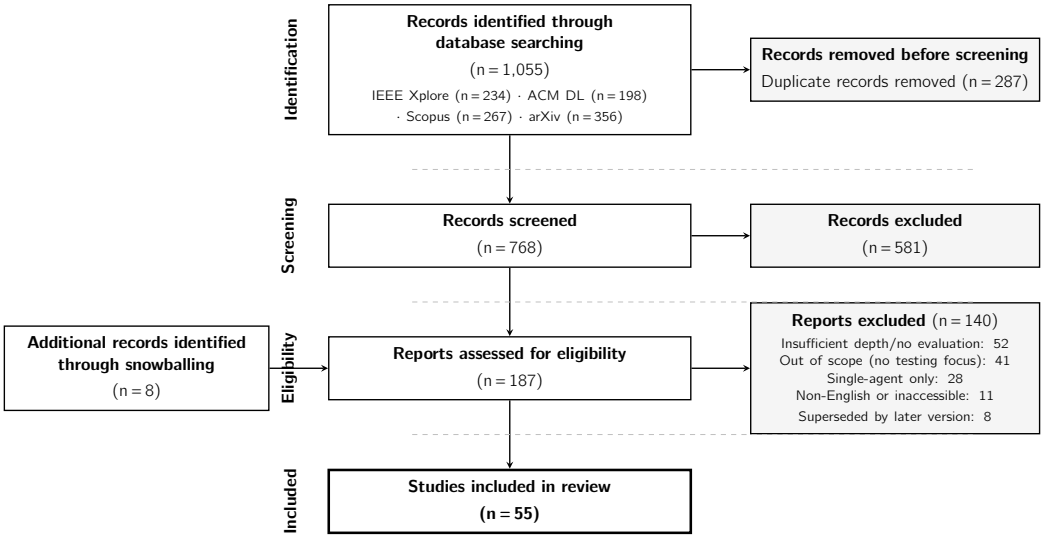

Figure 2.1: PRISMA 2020 flow diagram of the study selection process

The primary reasons for exclusion at the full-text stage are detailed in Table 2.7. Insufficient technical depth or lack of empirical evaluation was the most common reason, affecting 52 studies (37.1%), followed by out-of-scope content centred on pure code generation without testing relevance (41 studies, 29.3%), single-agent approaches without multi-agent coordination (28 studies, 20.0%), non-English or inaccessible full text (11 studies, 7.9%), and supersession by newer versions of the same work (8 studies, 5.7%).

Table 2.8 presents the quality assessment scores for representative studies from the corpus, illustrating the range of methodological rigour observed across the included works.

The 55 included studies exhibit a clear temporal concentration that mirrors the evolution of large language model capabilities, as shown in Table 2.9. Only four studies date from 2020, a period when LLM-based software engineering was still in its infancy and the dominant paradigm remained search-based test generation with tools such as EvoSuite (Fraser and Arcuri 2011). The count increased modestly to six studies in 2021, coinciding with the release of OpenAI Codex (Chen et al. 2021) and the first demonstrations that LLMs could generate syntactically correct test code from natural language descriptions. A more substantial rise occurred in 2022, with nine studies reflecting growing interest in code-specialised models and the publication of foundational benchmarks such as HumanEval.

Table 2.5: PRISMA study selection summary

Stage	Records
Identification	
Records from IEEE Xplore	234
Records from ACM Digital Library	198
Records from Scopus	267
Records from arXiv	356
Total records identified	1055
Duplicates removed	(287)
Screening	
Records screened (title/abstract)	768
Records excluded	(581)
Eligibility	
Full-text articles assessed	187
Full-text articles excluded	(140)
Articles from snowballing	8
Included	
Studies included in review	**55**

The sharpest acceleration appeared in 2023, which accounts for 19 of the 55 studies, over a third of the corpus. This surge followed the release of GPT-4, the publication of influential multi-agent frameworks including MetaGPT (Hong et al. 2023) and ChatDev (Qian et al. 2024), and the establishment of SWE-bench (Jimenez et al. 2024) as a standard evaluation benchmark for autonomous software engineering. The year 2024 contributed 13 studies, with a noticeable shift toward empirical evaluation and practical deployment concerns rather than purely architectural proposals. Four studies from early 2025 were captured through the arXiv preprint search, indicating sustained and growing research activity. This temporal distribution confirms that MAS-based test generation is a rapidly maturing field, with the bulk of the evidence base produced within the last three years.

The included studies appeared across a diverse range of publication venues, reflecting the interdisciplinary nature of LLM-based test generation research (Table 2.10). Premier software engineering conferences, including ICSE, FSE, and ASE, account for the largest share, contributing 16 studies that tend to emphasise architectural design and empirical evaluation on established benchmarks. Testing-focused venues such as ISSTA and ICST contributed seven studies, typically presenting more detailed evaluations of test quality metrics including mutation scores and fault detection rates. Seven studies appeared in AI and machine learning conferences (NeurIPS, ICLR), reflecting the dual identity of this research at the intersection of software engineering and artificial intelligence. Journals accounted for nine studies, published in outlets such as IEEE Transactions on Software Engineering (TSE), ACM Transactions on Software Engineering and Methodology (TOSEM), and Empirical Software Engineering (ESE). These journal publications generally offer more mature and thoroughly evaluated contributions compared to conference papers. Security-specific venues contributed three studies, addressing adversarial risks and privacy concerns. A notable proportion of the corpus, 13 studies representing nearly a quarter, appeared as arXiv preprints without formal peer review. While this raises questions about methodological rigour, it also

Table 2.6: Detailed search results by database and query

Database	Search String	Results	After Dedup
IEEE Xplore	S1 (MAS + Testing)	102	97
	S2–S4 (Few-Shot, Black-Box, Memory)	78	68
	S5 (Architecture + Deployment)	54	46
ACM DL	S1 (MAS + Testing)	86	79
	S2–S4 (Few-Shot, Black-Box, Memory)	68	56
	S5 (Architecture + Deployment)	44	37
Scopus	S1 (MAS + Testing)	112	94
	S2–S4 (Few-Shot, Black-Box, Memory)	93	74
	S5 (Architecture + Deployment)	62	49
arXiv	S1 (MAS + Testing)	148	121
	S2–S4 (Few-Shot, Black-Box, Memory)	126	97
	S5 (Architecture + Deployment)	82	50
Total		**1055**	**768**

Table 2.7: Full-text exclusion reasons

Exclusion Reason	Count	Percentage
Insufficient technical depth or no empirical evaluation	52	37.1%
Out of scope (pure code generation, not testing-related)	41	29.3%
Single-agent only (no MAS elements)	28	20.0%
Non-English or inaccessible full text	11	7.9%
Duplicate/superseded version	8	5.7%
Total Excluded	**140**	**100%**

captures the most recent developments in a field where the pace of innovation frequently outstrips the traditional publication cycle. The quality assessment scores described above were used to weight the contribution of preprints appropriately in the synthesis.

The majority of included studies address multiple research questions, which is expected given the interconnected nature of the research challenges (Table 2.11). RQ3 (multi-agent architectures) receives the broadest coverage, with 31 of the 55 studies discussing some form of agent coordination, role specialisation, or inter-agent communication. This prominence reflects the widespread adoption of multi-agent patterns in recent LLM-based software engineering tools and the growing consensus that single-agent approaches are insufficient for complex tasks such as comprehensive test generation. RQ1 and RQ2, golden examples and few-shot approaches (22 studies) and black-box testing and traffic analysis (16 studies), together represent the two input modalities central to this dissertation. The relatively lower count for black-box approaches suggests that traffic-based test generation remains a less explored direction compared to code-level techniques, which motivates the observer pipeline developed in this work. RQ5, covering practical deployment considerations including privacy, cost, and local execution, is addressed by 24 studies, indicating growing awareness of real-world adoption barriers. However, RQ4, persistent agent memory, is covered by only nine studies, making it the least explored area in the literature. This gap is particularly significant for this dissertation, as progressive learning through persistent memory represents one of

Table 2.8: Quality scores for selected studies

Study	QC1	QC2	QC3	QC4	QC5	QC6	QC7	Total
Hong et al. (MetaGPT)	1	1	1	1	0.5	0.5	1	6.0
Qian et al. (ChatDev)	1	1	1	1	0.5	0.5	1	6.0
Yang et al. (SWE-agent)	1	1	1	1	1	1	1	7.0
Wu et al. (AutoGen)	1	1	0.5	0.5	0.5	0.5	1	5.0
Jimenez et al. (SWE-bench)	1	1	1	1	1	1	1	7.0
Chen et al. (Codex)	1	1	1	1	1	0.5	1	6.5
Lemieux et al. (CODAMOSA)	1	1	1	1	1	1	1	7.0
Fraser & Arcuri (EvoSuite)	1	1	1	1	1	1	1	7.0
Greshake et al. (Prompt Inj.)	1	1	1	0.5	0.5	1	0.5	5.5

Table 2.9: Distribution of studies by publication year

Year	Count	Percentage
2020	4	7.3%
2021	6	10.9%
2022	9	16.4%
2023	19	34.5%
2024	13	23.6%
2025	4	7.3%
Total	**55**	**100%**

the key architectural innovations of the TestForge platform.

2.2.2 RQ1, Golden Examples and Few-Shot Test Generation

This subsection addresses RQ1 by examining the literature on leveraging existing test examples and few-shot prompting strategies for LLM-based test generation. The core insight is that providing high-quality reference tests as in-context examples can guide LLMs to generate tests that follow project conventions and target non-trivial scenarios.

Few-shot prompting, providing exemplar input-output pairs within the LLM prompt, has emerged as a key technique for guiding test generation without fine-tuning. Bareiss et al. (2022) conducted one of the earliest studies, evaluating the Codex model on three code-related tasks including test oracle generation and test case generation. Their findings indicate that few-shot language models are "surprisingly effective" but require specifically designed prompts and suitable input selection. Ouédraogo, Kaboré, Tian, et al. (2024) performed a large-scale evaluation of four LLMs across five prompt engineering techniques (zero-shot, few-shot, chain-of-thought, and combinations thereof), analysing 216,300 tests for 690 Java classes. Their results show that few-shot prompting consistently improves test correctness over zero-shot approaches, though the magnitude varies by model and task complexity. Ouédraogo, Kaboré, Y. Li, et al. (2024) reinforced these findings in their comprehensive study comparing GPT-3.5, GPT-4, Mistral 7B, and Mixtral 8x7B against EvoSuite, evaluating five prompting techniques. While few-shot learning improves test reliability, hallucination-driven compilation failures remain a persistent challenge, with rates up to 86% for smaller models.

Table 2.10: Distribution of studies by venue type

Venue Type	Count	Percentage
SE Conferences (ICSE, FSE, ASE)	16	29.1%
Testing Venues (ISSTA, ICST)	7	12.7%
AI/ML Conferences (NeurIPS, ICLR, UIST)	7	12.7%
Security Venues (AISec, EMNLP)	3	5.5%
Journals (TSE, TOSEM, ESE, IST, TOIS)	9	16.4%
arXiv Preprints	13	23.6%
Total	**55**	**100%**

Table 2.11: Number of studies addressing each research question

RQ	Topic	Studies
RQ1	Golden Examples and Few-Shot Test Generation	22
RQ2	Black-Box Testing and Traffic Analysis	16
RQ3	Multi-Agent Architecture Patterns	31
RQ4	Persistent Agent Memory	9
RQ5	Practical Deployment Considerations	24

A critical challenge in few-shot test generation is selecting the most relevant examples. Nashid, Sintaha, and Mesbah (2023) addressed this with CEDAR, a retrieval-based prompt selection technique that automatically identifies code demonstrations similar to the current task. For test assertion generation, CEDAR outperforms existing task-specific and fine-tuned models by 333% and 11% respectively, achieving 76% exact match accuracy. This demonstrates that intelligent example selection, rather than random or static examples, significantly impacts generation quality.

Several studies combine mutation testing feedback with few-shot prompting. Dakhel et al. (2024) proposed MuTAP, which augments LLM prompts with surviving mutants to iteratively improve test quality. Using mutation feedback as additional context, MuTAP achieves a mutation score of 93.57% on synthetic buggy code, substantially outperforming baseline approaches. At industrial scale, Foster et al. (2025) described Meta's production system for mutation-guided LLM test generation, demonstrating that surviving mutants serve as effective feedback signals for iterative test improvement.

Kang, Yoon, and Yoo (2023) explored a related application: using LLMs with few-shot prompting to generate bug-reproducing tests from bug reports (LIBRO). On the Defects4J benchmark, LIBRO successfully reproduced 33% of studied bugs (251 out of 750), suggesting a correct reproducing test in first place for 149 bugs. This demonstrates that few-shot prompting can capture complex testing patterns beyond simple assertion generation.

While the above studies use manually selected or retrieved code examples as few-shot context, none systematically extract testing patterns, conventions, assertion styles, fixture patterns, and testing strategies, from existing test suites using AST analysis. This gap motivates the Golden Examples pipeline proposed in this thesis, which uses Tree-sitter to parse reference tests and extract a structured style guide for few-shot prompting.

2.2.3 RQ2, Black-Box Testing and Traffic Analysis

This subsection addresses RQ2 by examining the literature on automated black-box testing of APIs, with particular attention to techniques for API discovery, specification inference, and test generation from observed traffic.

Golmohammadi, M. Zhang, and Arcuri (2023) provide a comprehensive survey of 92 scientific articles on RESTful API testing (2009–2022), establishing a taxonomy of tools and techniques. The survey identifies three main approaches: specification-based testing (requiring OpenAPI/Swagger definitions), fuzzing (random input generation with feedback), and stateful testing (modelling inter-request dependencies). Most tools require an existing API specification as input, which many real-world APIs lack.

Atlidakis, Godefroid, and Polishchuk (2019) introduced RESTler, the first stateful REST API fuzzer, which analyses Swagger specifications to infer producer-consumer dependencies between API operations. RESTler generates sequences of requests where outputs from one request feed as inputs to subsequent requests, discovering 28 bugs in GitLab and multiple bugs in Azure and Office365 cloud services. Y. Liu et al. (2022) extended this approach with Morest, which builds a dynamically updating RESTful-service Property Graph (RPG) to model API behaviours. Morest successfully requests 152–232% more API operations, covers 26–103% more lines of code, and detects 40–215% more bugs than prior techniques including RESTler.

Recent work has begun integrating LLMs into API testing. Kim, Stennett, et al. (2024) proposed RESTGPT, which uses LLMs to extract machine-interpretable rules and generate realistic parameter values from natural-language descriptions in API specifications. RESTGPT generates valid inputs for 73% of parameters versus only 17% for existing approaches, demonstrating that LLMs can bridge the gap between informal API documentation and formal test inputs. Kim, Sinha, and Orso (2025) developed LlamaRestTest, fine-tuning and quantising Llama3-8B for REST API testing with two components: inter-parameter dependency detection and realistic value generation. LlamaRestTest outperforms state-of-the-art tools (RESTler, Morest, EvoMaster) in code coverage and server error detection on 12 real-world services.

Most critically for our work, Decrop et al. (2024) proposed RESTSpecIT, the first approach that infers API documentation *and* performs black-box testing using LLMs *without requiring an existing specification*. Given only an API name and LLM access, RESTSpecIT generates request seeds and mutates them, achieving 88.62% route discovery and 89.25% query parameter discovery. This demonstrates the feasibility of LLM-driven black-box API testing, a core premise of the Observer pipeline proposed in this thesis.

While the above tools focus on fuzzing (finding crashes and security issues), none generate *human-readable test suites* from observed traffic. The combination of HTTP traffic capture (via proxy interception or HAR file import) with LLM-driven generation of pytest-style test suites is unexplored. This gap motivates the Black-Box Observer pipeline, which captures real API interactions and generates test code that verifies the observed behaviour.

2.2.4 RQ3, Multi-Agent Systems for Software Testing

This subsection addresses RQ3 by examining the architectural patterns and design principles employed in LLM-based multi-agent testing systems. The analysis reveals a rich design

space with significant variation in agent coordination models, role specialisation, and communication patterns.

The application of agent-based approaches to software engineering predates the LLM era. Traditional Multi-Agent System (MAS) research established foundational concepts including agent autonomy, social ability, reactivity, and pro-activeness (Wooldridge 2009). Early multi-agent software engineering tools focused on distributed development, collaborative editing, and automated code review, but lacked the natural language understanding capabilities that modern LLMs provide. The emergence of large language models, beginning with GPT-3 and accelerating with code-specialized models like Codex (Chen et al. 2021) and Code Llama (Rozière et al. 2023), enabled a new generation of agent systems capable of understanding and generating code with human-like proficiency. This capability, combined with reasoning techniques such as Chain-of-Thought (CoT) prompting (Wei et al. 2022) and ReAct (Yao et al. 2023), allows modern agents to tackle complex software engineering tasks that require multi-step reasoning and tool use.

The reviewed studies employ three primary coordination models, each with distinct characteristics and trade-offs. Hierarchical models establish a clear chain of command, with a central orchestrator agent decomposing tasks and delegating to specialized worker agents. MetaGPT (Hong et al. 2023) exemplifies this approach, implementing a "software company" metaphor where a Product Manager agent decomposes requirements, an Architect agent designs solutions, and Engineer agents implement and test code. Hierarchical coordination offers clear task decomposition and responsibility assignment, reduced communication overhead through centralized control, natural alignment with traditional software development roles, and simplified debugging through traceable decision chains. However, this model also introduces limitations, notably single points of failure at the orchestrator level and potential bottlenecks when the central agent must process all inter-agent communication. Peer-to-peer models allow agents to communicate directly without central mediation. ChatDev (Qian et al. 2024) implements a "chat chain" where agents engage in structured dialogues, with each agent both producing artifacts and reviewing work from peers. This model offers greater resilience through distributed decision-making, richer information exchange through direct agent interaction, emergent behaviours arising from agent collaboration, and a more natural representation of human team dynamics. At the same time, peer-to-peer coordination introduces challenges such as increased communication complexity and the potential for unproductive agent loops when appropriate termination conditions are not enforced. Many practical systems combine hierarchical and peer-to-peer elements. AutoGen (Wu et al. 2023) provides a flexible framework supporting both coordinator-worker patterns and direct agent conversations, allowing developers to configure coordination strategies appropriate to their specific use cases.

The literature reveals consistent patterns in how testing-related roles are distributed among agents. A planning or requirements agent typically initiates the testing workflow by analyzing requirements, identifying test objectives, and decomposing the testing task into manageable subtasks. This agent often employs structured output formats (e.g., JSON schemas) to ensure downstream agents receive well-defined specifications. Code analysis agents examine the System Under Test (SUT) to understand its structure, identify testable units, extract relevant context, and determine appropriate testing strategies. These agents may employ static analysis tools, parse abstract syntax trees, or analyze code semantics using LLM comprehension capabilities. The test generation agent produces test code based on inputs from planning and code analysis agents. Specialization may occur along multiple dimensions,

including test type, programming language, testing framework, and generation technique. Execution agents run generated tests in controlled environments, capture results, and report outcomes. These agents typically interact with external tools (test runners, containers, CI systems) through the ACI, making them critical from a security perspective. Validation or review agents assess the quality of generated tests, checking for issues such as trivial assertions that always pass, missing edge case coverage, test code that duplicates rather than validates SUT logic, flaky tests with non-deterministic behaviour, and style or maintainability concerns. When tests fail, debugging agents analyze failures to determine root causes, distinguishing between SUT bugs (true positives) and test defects (false positives).

Agent communication employs various protocols with different characteristics. In direct message passing, agents exchange structured messages containing task specifications, results, and feedback. Message formats range from natural language descriptions to strictly typed JSON schemas. Blackboard systems employ shared workspaces where agents post intermediate results and read inputs from a common data store. This approach decouples agents and simplifies adding new capabilities. In artifact-centric communication, rather than explicit messages, agents communicate through shared artifacts (code files, test suites, execution logs), with each agent modifying or extending artifacts produced by predecessors.

Tables 2.12 and 2.13 compare major multi-agent frameworks relevant to software testing, presenting both a high-level overview and a detailed comparison across key architectural dimensions.

Table 2.12: Comparison of multi-agent frameworks for software engineering

Framework	**Coordination**	**Roles**	**Testing Focus**	**Open Source**
MetaGPT	Hierarchical	Software company	Integrated	Yes
ChatDev	Peer-to-peer	Chat chain	Integrated	Yes
SWE-agent	Single + tools	Issue resolver	Bug fixing	Yes
AutoGen	Configurable	Flexible	General	Yes

Table 2.13: Detailed MAS framework comparison

Aspect	**MetaGPT**	**ChatDev**	**SWE-agent**	**AutoGen**
Coordination	Hierarchical SOPs	Chat chain	Single + tools	Configurable
Agent Count	5–7	4–6	1 (+ tools)	Variable
Primary Roles	PM, Architect, Engineer, QA	CEO, CTO, Programmer, Tester	Resolver	User-defined
Communication	Structured docs	Natural dialog	Tool calls	Messages
Memory	Shared artifacts	Chat history	State files	Conversation
Testing Support	Integrated	Integrated	Bug fixing	General
Open Source	Yes (MIT)	Yes (Apache)	Yes (MIT)	Yes (MIT)

The Agent-Computer Interface (ACI) defines how agents interact with external systems, including code repositories, execution environments, and development tools. SWE-agent (Yang et al. 2024) introduced principled ACI design, demonstrating that interface design

significantly impacts agent performance beyond LLM capability alone. Key ACI design considerations span several dimensions: the action space (what operations agents can perform), the observation space (what feedback agents receive after each action), state management (how context is maintained across interactions), error handling (how agents recover from failed operations), and permission boundaries (what access controls limit agent capabilities). Well-designed ACIs balance agent autonomy, enabling effective task completion, with safety constraints that prevent unintended or malicious actions.

2.2.5 RQ4, Persistent Agent Memory

This subsection addresses RQ4 by examining the literature on memory architectures for LLM-based agents, focusing on approaches that enable agents to accumulate and leverage knowledge across sessions.

Sumers et al. (2023) proposed CoALA (Cognitive Architectures for Language Agents), a framework drawing on cognitive science to organise agents along three dimensions: memory (working memory and long-term procedural/semantic/episodic memory), action space (internal reasoning and external grounding), and decision-making processes. CoALA provides a principled vocabulary for describing how agents store, retrieve, and use knowledge. Z. Zhang et al. (2025) conducted a comprehensive survey reviewing memory mechanisms in LLM-based agents, proposing a taxonomy of memory types (sensory, short-term, long-term) and memory operations (reading, writing, management). The survey identifies that while memory systems for conversational agents are well-studied, their application to task-specific domains such as software testing remains largely unexplored.

Packer et al. (2023) introduced MemGPT, which applies virtual context management inspired by operating system memory hierarchies to LLM agents. MemGPT divides memory into *core memory* (analogous to RAM, always in the context window), *archival memory* (persistent storage for long-term knowledge), and *recall memory* (conversation history). Agents manage memory through self-directed operations, deciding when to save, retrieve, or update information. MemGPT demonstrates effective multi-session conversation where agents maintain consistency across interactions, a capability directly relevant to progressive test generation. Shinn et al. (2023) introduced Reflexion, where agents verbally reflect on task feedback and maintain reflective text in an episodic memory buffer. Rather than updating model weights, Reflexion uses linguistic feedback as persistent memory for self-improvement across trials. Park et al. (2023) demonstrated generative agents with memory architectures that store experiences in natural language, synthesise higher-level reflections, and retrieve them dynamically for behaviour planning.

While memory architectures have been proposed for conversational agents, game-playing agents, and general-purpose assistants, their application to software testing, where agents accumulate knowledge about target applications (discovered endpoints, coverage gaps, learned patterns, failed assertions), has not been investigated. This thesis explores whether Letta-based persistent memory enables progressive improvement in test generation quality across sessions.

2.2.6 RQ5, Practical Deployment Considerations

This subsection addresses RQ5 by examining the practical considerations for deploying AI-based test generation systems, including model selection and configuration, evaluation approaches and benchmarks, security and privacy risks, mitigation strategies, industrial deployment patterns, and regulatory compliance. The analysis draws on 24 studies that address deployment-related concerns.

The selection of underlying models involves trade-offs across multiple dimensions including capability, cost, latency, privacy, and control. Proprietary models such as GPT-4, Claude, and Gemini offer state-of-the-art capabilities, particularly for complex reasoning tasks. Studies consistently report higher performance on challenging benchmarks when using frontier proprietary models. Proprietary models offer superior performance on complex tasks, continuous improvement through provider updates, robust API infrastructure with high availability, and advanced features such as function calling and structured outputs. However, they also present significant drawbacks: data must be transmitted to external providers raising privacy concerns, per-token costs scale with usage, organizations become dependent on provider availability and pricing, control over model behaviour is limited, and there is potential for training data contamination with widely used benchmarks. Open-weight models including LLaMA, Code Llama (Rozière et al. 2023), and StarCoder (R. Li et al. 2023) can be deployed locally, addressing privacy concerns and enabling customization through fine-tuning. Open-weight models present a compelling alternative: local deployment keeps data entirely on-premises, there are no per-token API costs beyond infrastructure, organizations retain full control over deployment and behaviour, the models can be fine-tuned for specific domains, and their architecture and training methodology are transparent. On the other hand, open-weight models generally exhibit lower capability than frontier proprietary models, require significant infrastructure investment for deployment, demand expertise in optimization and maintenance, and follow a slower improvement cycle than their proprietary counterparts.

Models trained specifically on code demonstrate superior performance on programming tasks compared to general-purpose models of similar size. Codex and GPT-4 are OpenAI's code-capable models, with GPT-4 representing the current capability frontier for code generation and understanding (Chen et al. 2021). Code Llama is Meta's code-specialized variant of LLaMA, available in 7B, 13B, and 34B parameter versions, with a Python-specialized variant (Rozière et al. 2023). StarCoder is a 15.5B parameter model trained on permissively licensed code, notable for its transparent training data and strong performance (R. Li et al. 2023). CodeBERT is an encoder-only model effective for code understanding tasks such as clone detection and defect prediction (Feng et al. 2020).

Studies reveal a complex trade-off between fine-tuning specialized models and engineering effective prompts for general-purpose models. Fine-tuning approaches adapt pre-trained models to specific testing tasks using domain-specific training data. Research has demonstrated that fine-tuned models can significantly outperform zero-shot prompting for specific testing tasks. Fine-tuning requires high-quality training data in the form of test-code pairs, substantial computational resources, expertise in model training and evaluation, and ongoing maintenance as project requirements evolve. Prompt engineering designs input formats that elicit desired behaviour from general-purpose models without modifying model weights. The most prominent techniques include few-shot prompting (providing examples of desired input-output pairs), chain-of-thought prompting (encouraging step-by-step reasoning) (Wei et al. 2022), ReAct (interleaving reasoning and action) (Yao et al. 2023), self-consistency

(sampling multiple outputs and selecting the answer by consensus), and tree-of-thoughts (exploring multiple reasoning paths before committing to a solution).

LLMs operate within fixed context windows constraining the information available for each inference. Managing context effectively is crucial for testing tasks that may involve large codebases. Several strategies have been proposed to address this constraint. Retrieval-Augmented Generation (RAG) retrieves code snippets relevant to the current task from an indexed codebase. Hierarchical summarization compresses code into summaries at multiple abstraction levels, preserving essential information while reducing token consumption. Selective context inclusion restricts the prompt to immediately relevant code and documentation, while sliding window approaches process large codebases in overlapping chunks to maintain continuity across segments. Several key configuration parameters also affect generation behaviour. Temperature controls the degree of randomness, with lower values producing more deterministic outputs. Top-p (nucleus sampling) limits the sampling pool to the highest-probability tokens. Maximum token limits constrain output length, while stop sequences define generation termination conditions. For test generation, studies suggest moderate temperatures in the range of 0.2–0.4, balancing creativity with consistency, though optimal settings vary by task.

The literature employs diverse metrics and benchmarks to evaluate testing effectiveness, providing the empirical foundation for comparing MAS-based approaches to traditional methods. Coverage metrics measure the extent to which generated tests exercise the SUT, with the most commonly reported being line coverage, branch coverage, method coverage, and mutation score. Correctness metrics assess whether generated tests are valid and meaningful, including the compilation rate, execution rate, pass rate, and assertion validity. Bug detection metrics evaluate the ability to identify real defects through the true positive rate, false positive rate, and bug detection efficiency. The pass@k metric, widely used in code generation evaluation, measures the probability of generating at least one correct solution within k attempts (Chen et al. 2021):

$$\text{pass@k} = \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{2.1}$$

where n is the total number of samples and c is the number of correct samples.

Key benchmark datasets include HumanEval (Chen et al. 2021), containing 164 hand-written Python programming problems; the Mostly Basic Python Problems (MBPP) benchmark (Austin et al. 2021), providing 974 crowd-sourced Python tasks; SWE-bench (Jimenez et al. 2024), which evaluates agents on real GitHub issues from popular Python repositories; and Defects4J (Just, Jalali, and Ernst 2014), a database of real bugs from open-source Java projects. Studies consistently demonstrate that multi-agent architectures outperform single-agent approaches on complex testing tasks. MetaGPT (Hong et al. 2023) reported significantly higher pass rates on software development tasks compared to single-agent baselines, with testing quality directly impacting overall success rates. Comparisons between LLM-based and traditional testing tools reveal complementary strengths. EvoSuite (Fraser and Arcuri 2011) achieves high branch coverage through search-based generation but produces tests that are often difficult to understand and maintain. Randoop (Pacheco et al. 2007) generates many tests quickly through random exploration but with limited semantic awareness. Pynguin (Lukasczyk, Kroiß, and Fraser 2023) extends search-based test generation to Python with multiple algorithms (DynaMOSA, MIO, MOSA), providing a key baseline

for comparing LLM-based Python test generation. LLM-based tools, by contrast, generate more readable and semantically meaningful tests but may miss edge cases that systematic approaches find. Schäfer et al. (2024) (TestPilot) achieved 70.2% median statement coverage on npm packages, significantly outperforming traditional tools, while Z. Yuan et al. (2024) (ChatTester) found that only 24.8% of vanilla ChatGPT-generated tests pass execution, though the passing tests achieve comparable coverage to manually-written ones. CODAMOSA (Lemieux et al. 2023) proposed combining LLM generation with search-based testing to escape coverage plateaus, achieving higher coverage than either approach alone. Pizzorno and Berger (2024) (CoverUp) extended this idea with coverage-guided prompting, achieving 80% median line+branch coverage versus 47% for CODAMOSA. Mutation testing (Jia and Harman 2011) provides a rigorous framework for evaluating test quality by measuring the proportion of artificial faults (mutants) detected by the test suite, and LLM-generated tests have shown promising results on mutation testing benchmarks, particularly when guided by mutation analysis feedback. Several factors complicate interpretation of reported effectiveness results, including benchmark contamination, incompatible evaluation metrics across studies, limited reproducibility due to proprietary models, and task selection bias in benchmarks.

The security and privacy risks associated with LLM-based multi-agent testing systems constitute key practical considerations for deployment. The analysis identifies five primary risk categories: data leakage, adversarial manipulation, unsafe code generation, grounding failures, and ACI vulnerabilities (Table 2.14). Data leakage occurs when sensitive information is inadvertently exposed through agent interactions with LLM providers or external systems. Testing agents require access to source code, which may contain proprietary algorithms, trade secrets, or competitive intelligence. When using cloud-based LLM providers, this code is transmitted to external servers, potentially violating confidentiality requirements. Codebases frequently contain Personally Identifiable Information (PII) in various forms, including test fixtures with realistic user data, configuration files with credentials or API keys, and database seeds with sample customer information. Agents processing such codebases may transmit PII to external providers, potentially violating GDPR requirements regarding data transfers and purpose limitation. System prompts defining agent behaviour may themselves contain sensitive information, and prompt injection attacks may extract these system prompts, exposing protected information.

Adversarial actors may exploit agent systems through various manipulation techniques. Prompt injection attacks embed malicious instructions in data processed by agents (Greshake et al. 2023). In testing contexts, injection vectors are particularly diverse: malicious instructions may be hidden in code comments, embedded in function docstrings, inserted as string literals, encoded in specially crafted file names, or planted in error messages that agents subsequently analyse. Trojan comments or code patterns may be planted in repositories to trigger specific agent behaviors when encountered. Multi-agent systems may exhibit emergent behaviors not intended by designers, particularly when agents pursue local objectives that conflict with global system goals. Agents generating test code may introduce security vulnerabilities or malicious functionality, including SQL injection vulnerabilities in database test helpers, path traversal issues in file handling tests, command injection flaws, insecure deserialization in test fixtures, or hardcoded credentials. LLMs may hallucinate package names that do not exist, creating opportunities for "slopsquatting" attacks, and may generate slightly misspelled package names that attackers have registered as typosquatting packages. The "grounding gap" between LLM knowledge and the actual state of the system under test can cause agents to generate tests calling methods or APIs that do not exist,

suggest deprecated patterns, or generate tests inconsistent with code outside the visible context window. The Agent-Computer Interface represents a critical attack surface, with risks including excessive permissions, insufficient isolation, and audit trail gaps.

Table 2.14: Comprehensive security threat catalog

ID	Threat Category	Specific Threat	Studies
T1	Data Leakage	Source code exfiltration	S03, S12, S27
T2	Data Leakage	PII exposure via prompts	S03, S15, S31
T3	Data Leakage	Credential/secret exposure	S12, S19, S31
T4	Data Leakage	Prompt leakage attacks	S15, S22
T5	Adversarial	Direct prompt injection	S08, S15, S22
T6	Adversarial	Indirect prompt injection	S15, S22, S34
T7	Adversarial	Trojan code comments	S22, S34
T8	Adversarial	Agent alignment failures	S08, S27
T9	Code Generation	Vulnerability introduction	S19, S27, S38
T10	Code Generation	Dependency confusion	S19, S38
T11	Code Generation	Typosquatting packages	S19, S38
T12	Code Generation	Malicious code insertion	S27, S34
T13	Grounding	Hallucinated APIs	S05, S12, S27
T14	Grounding	Outdated knowledge	S05, S12
T15	Grounding	Context inconsistency	S12, S27
T16	ACI	Excessive permissions	S03, S08, S27
T17	ACI	Insufficient isolation	S03, S19, S27
T18	ACI	Audit trail gaps	S03, S31

The literature proposes architectural patterns and mitigation strategies to address these risks, organized into three categories. Model-centric strategies address risks at the LLM layer itself: deploying open-weight models locally eliminates data transmission to external providers; models can be fine-tuned to reject harmful requests through RLHF and Constitutional AI approaches; and post-generation filtering can detect and block unsafe outputs before they reach downstream systems. Pipeline-centric strategies implement security controls in the infrastructure surrounding agents: test execution should occur in isolated environments (containers, virtual machines, serverless functions, or WebAssembly sandboxes); data preprocessing can remove or redact PII before code reaches agents; context minimization provides agents only the information strictly necessary for their tasks; and securing the ACI involves explicit permission models, action validation, rate limiting, allowlisting, and confirmation gates for high-risk operations. Comprehensive logging enables detection, forensics, and compliance. Algorithmic strategies use testing and validation techniques to verify agent behavior: mutation testing provides objective quality metrics for generated tests, combinatorial testing systematically varies inputs to identify failure conditions, and chaos engineering deliberately injects failures to verify system resilience. Effective security requires layering multiple mitigations across prevention, detection, response, and recovery.

Organizations typically begin with shadow deployments where agent-generated tests run alongside but do not replace existing test suites. MAS testing systems can integrate at various CI/CD stages, including pre-commit generation, pull request review, continuous testing on each commit, nightly comprehensive generation, and release gate validation. LLM costs

depend on input tokens, output tokens, model selection, and request frequency. The primary benefits include reduced time spent writing boilerplate test code, faster identification of missing coverage, earlier bug detection, and reduced cognitive load during test maintenance. Adoption requires prompt engineering expertise, security awareness, and the ability to critically evaluate agent outputs.

The regulatory landscape affecting deployment requires careful attention. The General Data Protection Regulation (GDPR) (European Parliament and Council 2016) imposes requirements on processing of personal data, including data minimisation (Article 5(1)(c)), purpose limitation (Article 5(1)(b)), restrictions on international data transfers (Chapter V), and Data Protection Impact Assessment requirements (Article 35). The EU AI Act (European Parliament and Council 2024) classifies AI systems into risk tiers and introduces transparency requirements (Article 52). Testing systems are likely classified as limited or minimal risk unless they test safety-critical systems. When agent-generated tests fail to detect bugs, liability questions arise regarding organizational responsibility, provider liability, and the impact of human review on liability allocation. Following the seven privacy-by-design principles (Cavoukian 2011), testing systems should adopt a proactive posture, make privacy the default setting, embed privacy into system design, achieve full functionality without unnecessary trade-offs, ensure end-to-end security, maintain visibility and transparency, and keep user privacy central to all design decisions.

2.3 Critical Discussion

This section presents a critical analysis of the findings reported above, evaluating the strengths and limitations of the existing literature for each review theme and identifying the research gaps that motivate the contributions of this thesis.

2.3.1 RQ1, On the Limits of Few-Shot Test Generation

The evidence for few-shot prompting is convincing but incomplete. The studies reviewed demonstrate that providing examples improves generation quality, yet the examples used are almost universally selected at the code-snippet level, a function and its corresponding test, rather than at the project level. Real-world testing involves conventions that span entire test suites: consistent fixture patterns, shared setup and teardown logic, project-specific assertion styles, and framework-specific idioms. None of the reviewed studies attempt to capture these higher-level patterns systematically. The reliance on surface-level code similarity for example retrieval, as in CEDAR, means that structural testing conventions, such as always verifying state after mutation or testing both positive and negative paths, are lost when examples are selected solely on syntactic resemblance.

Furthermore, the hallucination rates reported by Ouédraogo, Kaboré, Y. Li, et al. (2024) (up to 86% for smaller models) suggest that few-shot prompting alone is insufficient. A practical system must include an execution-validation loop that detects and discards non-compiling or failing tests. This motivates the multi-agent architecture adopted in TestForge, where a dedicated Executor agent runs generated tests and a Validator agent assesses their quality, creating a feedback loop that compensates for LLM hallucinations. The gap in AST-based pattern extraction, where Tree-sitter parsing could produce structured style guides capturing project-wide testing conventions, remains the primary opportunity that the Golden Examples Pipeline addresses.

2.3.2 RQ2, On the Untapped Potential of Traffic Observation

The black-box testing literature is heavily biased toward fuzzing, which optimises for crash detection and code coverage rather than the production of maintainable, human-readable test suites. While fuzzing tools such as RESTler and Morest are effective at finding server errors and security vulnerabilities, the tests they produce are machine-generated sequences of requests that bear little resemblance to what a developer would write. This is a critical limitation for organisations seeking to build regression test suites from observed behaviour: the output of a fuzzer cannot be checked into a repository and maintained alongside the application code.

The emergence of specification-free approaches like RESTSpecIT is encouraging, as it demonstrates that LLMs can infer API structure without formal documentation. However, the step from endpoint discovery to test suite generation, producing pytest functions with meaningful assertions, shared fixtures, and proper test isolation, has not been taken. The Observer Pipeline proposed in this thesis occupies precisely this gap: it combines traffic capture (via proxy interception or HAR import) with LLM-driven generation to produce structured, human-readable test code. The critical question, which the experimentation chapter addresses, is whether the information contained in captured HTTP exchanges provides sufficient context for the LLM to generate tests that go beyond trivial status code assertions.

2.3.3 RQ3, On Agent Architecture Trade-Offs

The proliferation of multi-agent frameworks is both a strength and a weakness of the current literature. On the positive side, the diversity of coordination models demonstrates that there is no single correct architecture: hierarchical, peer-to-peer, and hybrid approaches each have valid use cases. On the negative side, the lack of controlled comparisons between architectures makes it difficult to determine which patterns are most effective for specific tasks. Most framework papers evaluate their approach against single-agent baselines or entirely different systems, rather than comparing alternative multi-agent designs for the same task.

For test generation specifically, a pipeline architecture, where data flows sequentially through analysis, generation, execution, and validation stages, has clear advantages over more complex coordination models. Testing is inherently sequential: one cannot validate a test before generating it, and one cannot generate a test without first understanding the target application. The pipeline model also provides natural checkpoints for quality control and makes the system easier to debug and extend. This is the rationale behind the TestForge architecture, which adopts a pipeline coordination model with typed data artifacts flowing between specialised agents, rather than the more complex dialogue-based or hierarchical approaches favoured by general-purpose frameworks.

A further observation is that the ACI design insights from SWE-agent are underappreciated in the testing literature. Most multi-agent testing papers focus on agent roles and communication patterns but pay little attention to how agents interact with external tools, test runners, build systems, version control. Yet as Yang et al. (2024) demonstrated, interface design can impact agent performance as much as or more than the underlying LLM capability. TestForge addresses this by providing each agent with a carefully scoped interface: the Executor agent interacts only with pytest through a sandboxed subprocess, the Observer agent interacts only with HTTP traffic data, and so forth.

2.3.4 RQ4, On the Absence of Memory in Testing Systems

The near-total absence of persistent memory in testing-oriented agent systems represents what is perhaps the most significant gap in the literature. Software testing is fundamentally a progressive activity: as a developer tests an application over days and weeks, they accumulate knowledge about its behaviour, its failure modes, and its most fragile components. Current LLM-based testing tools discard this knowledge after every session, forcing the system to rediscover the same endpoints, re-learn the same conventions, and potentially regenerate the same tests.

MemGPT's virtual context management provides a technically sound foundation for addressing this gap, but it was designed for conversational agents, not task-oriented testing workflows. The challenge for testing-specific memory lies in determining what to remember: discovered endpoint maps, successful test patterns, previously found bugs, coverage gaps, and application-specific conventions all represent potentially valuable persistent knowledge, but storing everything risks overwhelming the retrieval mechanism with irrelevant information. TestForge explores this through a ContextStore that maintains per-application memory, combined with Letta-based agent memory for conversational interaction. The experimentation chapter evaluates whether this architecture delivers measurable improvement across sessions.

2.3.5 RQ5, On the Privacy-Capability Trade-Off

The practical deployment literature consistently frames local execution as a privacy solution with a capability cost, but this framing may be overly pessimistic. The rapid improvement in open-weight models, from Code Llama 7B through to recent instruction-tuned variants, is narrowing the gap with proprietary models for code-specific tasks. For test generation in particular, the reasoning demands may be lower than for general-purpose software engineering: the LLM must understand an API structure and produce well-formed test functions, but it does not need to reason about novel algorithms or complex system architectures.

The more pressing concern for local deployment is infrastructure cost and operational complexity. Running a quantised 8B parameter model on consumer hardware is feasible but slow, with inference times of 30–60 seconds per generation that would be unacceptable in an interactive workflow. TestForge addresses this by adopting a batch-oriented design where the user initiates a generation run and reviews results asynchronously, and by using LiteLLM as a unified gateway that allows seamless switching between local and cloud models depending on the sensitivity of the target codebase. The regulatory analysis confirms that for organisations subject to GDPR Article 44–49 on international data transfers, local deployment is not merely a preference but a practical necessity when processing codebases that contain personal data.

2.3.6 Research Gaps

The critical analysis above identifies five concrete research gaps that this thesis addresses. First, no existing approach extracts project-wide testing patterns from reference test suites using AST analysis for structured few-shot prompting. Second, no tool generates human-readable, maintainable test suites from observed HTTP traffic without requiring API specifications or source code access. Third, no multi-agent testing architecture combines multiple complementary input strategies, golden examples and black-box observation, within a unified platform. Fourth, persistent memory for progressive improvement in test generation has

not been investigated. Fifth, a systematic evaluation of the quality trade-offs, performance characteristics, and cost-effectiveness of fully local test generation using open-weight models has not been conducted.

2.3.7 Implications for the Proposed Research

The identified gaps directly motivate the contributions proposed in this thesis. The limited exploration of example-driven approaches for test generation motivates the Golden Examples Pipeline, which leverages existing test suites as structured few-shot examples to generate insightful tests that go beyond trivial assertions. The absence of black-box testing approaches that combine traffic analysis with LLM-driven generation motivates the Black-Box Observer Pipeline, which captures HTTP traffic to automatically map endpoints and generate test suites without requiring source code access. The lack of integrated multi-agent testing architectures that support multiple complementary input strategies motivates the development of TestForge as a modular platform with six specialised agents covering the full testing workflow from analysis through validation. The underdeveloped area of persistent agent memory for testing motivates the integration of Letta-based memory, enabling the platform to accumulate knowledge about target applications and improve progressively across sessions. Finally, the need for practical deployment guidance under privacy constraints motivates the local execution assessment, which evaluates the feasibility of running the complete platform on local infrastructure using open-weight models.

2.4 Conclusions of the Review

This systematic literature review examined 55 studies following the PRISMA 2020 methodology, organised around five research questions that collectively address the overarching review question of how AI techniques have been applied to automated software test generation. The review covered publications from 2020 to early 2026 across four major databases, selecting studies through a rigorous multi-stage screening process with explicit inclusion and exclusion criteria.

The Results section consolidated the evidence for each research question. For RQ1, the literature demonstrates that few-shot prompting consistently improves test generation quality over zero-shot approaches, but no existing work systematically extracts project-level testing patterns from reference test suites using AST analysis. For RQ2, black-box API testing is dominated by fuzzing-oriented approaches that optimise for crash detection rather than the production of human-readable, maintainable test suites from observed traffic. For RQ3, multi-agent architectures consistently outperform single-agent approaches, with pipeline, hierarchical, and peer-to-peer coordination models each offering distinct trade-offs, though controlled comparisons between architectures for the same testing task are absent. For RQ4, persistent memory mechanisms have been proposed for conversational and general-purpose agents but have not been applied to software testing, leaving the potential for progressive learning in test generation entirely unexplored. For RQ5, the tension between model capability and deployment constraints is well-characterised, with the regulatory landscape strongly favouring local execution for organisations handling sensitive data, though systematic evaluations of local deployment feasibility remain limited.

The Critical Discussion identified five concrete research gaps: the absence of AST-based pattern extraction for structured few-shot prompting, the lack of traffic-to-test-suite generation tools, the need for dual-pipeline architectures combining golden examples with black-box

observation, the unexplored potential of persistent memory for testing agents, and the limited evaluation of fully local test generation platforms. These gaps directly motivate the design of the TestForge platform presented in the subsequent chapters.

2.4.1 Limitations of This Review

This review has several limitations that should be acknowledged. The field is evolving rapidly, and some relevant work may have been published after the search cutoff date of January 2026. Publication bias may cause the included studies to overrepresent positive results, with failed approaches and negative findings going underreported in the academic literature. Industry practices that are not disseminated through academic venues constitute a grey literature gap that may leave practical deployment insights underrepresented in the synthesis. The restriction to English-language publications may exclude relevant work published in other languages, particularly from research communities in China, where significant LLM development is taking place. Finally, many of the reviewed studies rely on benchmarks such as HumanEval and SWE-bench that are potentially contaminated by LLM training data, which may inflate the effectiveness claims reported in the literature.

2.4.2 Future Directions

The systematic review identifies several directions for future research that extend beyond the scope of this thesis but represent significant opportunities for the field.

The first direction concerns the integration of formal verification techniques with LLM-based test generation. While the reviewed studies focus on generating tests through prompting and example-driven approaches, combining these methods with formal methods, such as property-based testing or model checking, could provide stronger guarantees about test correctness and coverage completeness. This integration remains largely unexplored in the multi-agent context.

The second direction involves cross-language and cross-framework generalisation. The majority of the reviewed studies target a single programming language (predominantly Python or Java) and a single testing framework. Investigating how multi-agent testing systems can transfer knowledge across languages, frameworks, and application domains would significantly broaden their practical applicability.

A third direction concerns human-agent collaboration models. The reviewed literature treats test generation as a largely autonomous process, with human involvement limited to reviewing outputs. Research into interactive workflows, where developers and agents collaboratively refine test strategies, share domain knowledge, and iteratively improve test suites, could yield higher-quality outcomes than fully autonomous approaches.

The fourth direction addresses longitudinal evaluation of persistent memory. While this thesis evaluates memory-augmented testing across a limited number of sessions, long-term studies tracking how agent knowledge evolves over months of continuous use on production codebases would provide stronger evidence for the value of persistent memory in practice.

Finally, the review identifies a need for standardised evaluation benchmarks specific to AI-driven test generation. Current evaluations rely on a patchwork of benchmarks designed for code generation (HumanEval), bug fixing (SWE-bench, Defects4J), or manual evaluation. A purpose-built benchmark capturing the full spectrum of test quality, including assertion

depth, test isolation, fixture design, and maintainability, would enable more rigorous and comparable evaluations across the field.

Chapter 3

Data Collection and Pre-processing

This chapter describes the data sources, collection methods, and pre-processing steps that feed the two pipelines of the TestForge platform. The Golden Examples pipeline relies on existing test files as its primary data source, while the Black-Box Observer pipeline captures HTTP traffic from running applications. Both pipelines transform raw inputs into structured representations suitable for LLM-based test generation.

3.1 Data Sources and Selection

3.1.1 Selection Criteria and Objectives

The quality of generated tests depends critically on the quality and representativeness of input data. For the Golden Examples pipeline, this means selecting reference test files that demonstrate good testing practices. For the Observer pipeline, it means capturing sufficiently diverse HTTP interactions to cover the application's API surface.

Several criteria guided data source selection for both pipelines. Input data should be representative, covering the range of testing patterns and API behaviours that the platform will encounter in practice. Golden test examples should demonstrate quality through meaningful assertions, proper fixture usage, and comprehensive scenario coverage rather than trivial smoke tests. Sources should offer diversity, spanning different testing styles, HTTP methods, error handling patterns, and application domains to prevent overfitting to a single pattern. Finally, traceability requires that each data source be documented with its origin, collection method, and any transformations applied.

3.1.2 Golden Test Examples

Golden examples are existing, human-written test files that serve as reference patterns for the generation process. These files encode implicit knowledge about testing framework conventions such as pytest fixtures, parametrize decorators, and assertion styles, as well as HTTP client usage patterns including requests library calls, authentication headers, and response parsing. They also capture test naming conventions and documentation practices, fixture design patterns covering setup/teardown semantics, dependency injection, and scope management, and assertion strategies ranging from status code checking through response body validation to error message verification.

For the evaluation presented in Chapter 6, golden examples were collected from a Flask CRUD API application comprising 23 test functions across 3 test files. These tests cover all CRUD operations (Create, Read, Update, Delete) and include both happy-path and error

scenarios. All tests make consistent use of pytest fixtures with the `requests` HTTP client, and include search endpoint testing with query parameter validation.

3.1.3 HTTP Traffic Captures

The Observer pipeline accepts HTTP traffic data from three sources: pre-captured exchange data in the form of JSON arrays of HTTP request/response pairs provided directly via the API, HAR (HTTP Archive) files (W3C Web Performance Working Group 2012) in the industry-standard format exported from browser developer tools or proxy servers, and live traffic capture through real-time interception via mitmproxy.

Each captured exchange includes: HTTP method, URL, path, query parameters, request headers, request body, response status code, response headers, response body, and timing information.

3.2 Pre-processing: Golden Examples Pipeline

3.2.1 AST Parsing with Tree-sitter

The Analyser Agent uses Tree-sitter, an incremental parsing library (Brunsfeld et al. 2018), to perform Abstract Syntax Tree (AST) analysis on golden test files. Tree-sitter was selected over Python's built-in `ast` module for its ability to handle partial or syntactically incomplete code gracefully, a practical consideration when processing real-world test files that may contain syntax variations.

The parsing process extracts the following structural elements from each test file:

Imports All import statements, distinguishing between standard library, third-party, and local imports. This information determines which libraries the generated tests should reference.

Fixtures Functions decorated with `@pytest.fixture`, including their names, parameter lists, scope declarations, and whether they use `yield` for setup/teardown semantics.

Test Functions Functions whose names begin with `test_`, including their docstrings, parameter dependencies (fixture injection), and the HTTP methods and endpoints they target.

Assertions All assertion statements within test functions, classified by type: status code checks (`assert response.status_code == ...`), response body validations, collection membership tests, and exception assertions.

Helper Functions Non-test, non-fixture functions that provide utility support (e.g., data factories, response parsers).

3.2.2 HTTP Endpoint Detection

A critical pre-processing step is extracting the HTTP methods and endpoints targeted by each test function. The analyser employs a multi-strategy approach. The first strategy, direct method call detection, identifies calls to `requests.get()`, `requests.post()`, `requests.put()`, `requests.delete()`, and `requests.patch()` within each test function body. The second strategy, f-string URL extraction, parses f-string arguments to HTTP

client calls (e.g., `f"{base_url}/api/users/{user_id}"`) using regular expressions to extract the path component after the base URL variable. The third strategy handles non-f-string URL arguments where the path is embedded as a plain string literal.

For each detected HTTP call, the analyser records the HTTP method (GET, POST, PUT, DELETE, PATCH), the endpoint path (e.g., `/api/users`, `/api/users/{id}`), and the position within the test function, distinguishing setup calls from assertion targets.

3.2.3 Pattern Aggregation into Style Guide

After parsing all golden examples, the Analyser Agent aggregates the extracted patterns into a `TestStyleGuide`, a structured summary of the testing conventions observed across the example files. The style guide captures the detected testing framework (e.g., `pytest`) and HTTP client library (e.g., `requests`), along with the test function naming pattern (e.g., `test_<action>_<scenario>`). It also records the set of common imports shared across multiple golden files, common fixtures that indicate reusable test infrastructure, and the most frequently used assertion forms with their relative frequencies. Finally, it includes average metrics such as assertions per test, test length, and the proportion of tests containing docstrings. This style guide serves as structured context for the Generator Agent's LLM prompt, enabling few-shot generation that aligns with the project's existing conventions.

3.3 Pre-processing: Observer Pipeline

3.3.1 HTTP Exchange Normalisation

Raw HTTP traffic data varies significantly in format depending on the capture method. The Observer Agent normalises all inputs into a common `HTTPExchange` representation through several transformations. Method normalisation uppercases HTTP methods and validates them against the standard set. URL decomposition splits full URLs into base URL, path, and query parameters. Body parsing processes request and response bodies as JSON where the content type indicates `application/json`, storing them as raw text otherwise. Header filtering removes headers irrelevant to API semantics (e.g., `Connection`, `Accept-Encoding`) while preserving authentication-related headers.

For HAR file input, the parser follows the W3C HTTP Archive specification, which contains detailed timing, cookie, and redirect information beyond what is needed for test generation. The HAR parser extracts only the fields relevant to API testing: it iterates over the `log.entries` array, extracting for each entry the `request.method`, `request.url`, `request.headers`, and `request.postData`, along with the `response.status`, `response.headers`, and `response.content.text`. Non-API requests (static assets, tracking pixels) are filtered out based on content type and URL patterns.

3.3.2 Endpoint Mapping and Schema Inference

The Mapper Agent transforms normalised HTTP exchanges into an `EndpointMap`, a structured representation of the application's API surface. This process involves four key operations.

Path normalisation detects dynamic path segments and replaces them with parameter placeholders. Numeric segments (e.g., `/users/42`) are replaced with `{id}`, and UUID

segments are replaced with `{uuid}`. The resulting normalised paths group exchanges that target the same logical endpoint.

Schema inference examines multiple exchanges to the same endpoint to infer JSON schemas for request and response bodies. It records field names, types, and whether fields appear consistently (required) or intermittently (optional), and detects common patterns such as pagination fields, error response structures, and nested objects.

Authentication pattern detection analyses request headers across all exchanges to identify authentication mechanisms. Bearer token authentication is detected via `Authorization: Bearer ...` headers, while basic authentication is identified through `Authorization: Basic ...` headers. API key authentication is recognised by common header names such as `X-API-Key` and `api-key`, and cookie-based sessions are detected through `Cookie` or `Set-Cookie` headers.

Dependency chain detection identifies relationships between endpoints by analysing the flow of identifiers. A `POST` endpoint that returns an `id` field is linked to subsequent `GET`, `PUT`, and `DELETE` endpoints that use the same identifier as a path parameter. These dependencies inform the test generation order, ensuring that tests create required resources before attempting to read, update, or delete them.

3.4 Data Representation Models

All data flowing through the TestForge pipelines is represented using Pydantic v2 models (Colvin 2017), which provide runtime type validation, serialisation, and documentation. The key data models are:

`GoldenExample` A parsed test file containing lists of imports, fixtures, test functions, helper functions, and class definitions. Each test function includes its name, docstring, HTTP method, endpoint, parameter list, and assertions.

`TestStyleGuide` An aggregated summary of testing conventions extracted from multiple golden examples, as described in Section 3.2.3.

`HTTPExchange` A single HTTP request/response pair with method, URL, headers, body, status code, and timing.

`EndpointInfo` A single API endpoint with its method, path, request/response schemas, authentication requirements, and sample exchanges.

`EndpointMap` The complete API surface of a target application: a collection of `EndpointInfo` objects with detected authentication patterns and endpoint dependencies.

`TestSuite` A collection of generated tests with metadata, quality scores, and a method for rendering the suite as executable Python code.

`AppContext` A persistent per-application record of discovered endpoints, coverage status, previous run results, and learned patterns, enabling progressive improvement across sessions.

3.5 Risk Analysis and Data Bias

3.5.1 Golden Example Bias

The quality and diversity of golden examples directly constrain the quality of generated tests. Several bias risks were identified. First, style overfitting may occur if golden examples use a narrow range of assertion patterns, causing the generator to produce tests limited to the same patterns and missing opportunities for more insightful test strategies. Second, coverage bias arises when golden examples disproportionately cover happy-path scenarios, leading to underrepresentation of error cases, boundary conditions, and security-related tests in generated output. Third, framework lock-in is a concern because the current implementation assumes pytest and the requests library; golden examples using other frameworks such as unittest or aiohttp would require adapter support.

3.5.2 Observer Data Limitations

HTTP traffic captures may not represent the full API surface. Incomplete coverage is a primary concern: if the observed session does not exercise all endpoints, the Mapper Agent will produce an incomplete endpoint map. Sampling bias compounds this problem, as traffic captures may overrepresent frequently-used endpoints and underrepresent administrative or edge-case operations. Additionally, state dependency poses a challenge because some API behaviours depend on prior state (e.g., deleting a resource requires first creating it), and a single observation session may not capture all state transitions.

3.5.3 Mitigation Strategies

To address these biases, the platform implements several countermeasures. The LLM prompt explicitly instructs the generator to produce tests in eight insightful categories (state integrity, boundary probing, business logic, error quality, concurrency hints, authorisation boundaries, data leakage, and regression traps) regardless of the patterns observed in golden examples. The persistent memory system (`AppContext`) tracks which endpoints have been tested and which remain untested, guiding subsequent generation runs toward coverage gaps. Furthermore, the combined pipeline mode allows golden examples and observer data to be used together, compensating for the limitations of each approach individually.

Chapter 4

Methods and Tools

This chapter presents the system architecture of TestForge, describing the dual-pipeline design, the multi-agent architecture, the LLM integration strategy, and the persistent memory mechanism. It also describes the development stack and deployment environment.

4.1 System Overview and Architecture

TestForge is organised around two complementary pipelines that can operate independently or in combination. The Golden Examples Pipeline analyses existing test files to learn patterns, then generates new tests in the same style while targeting untested scenarios. The Black-Box Observer Pipeline captures HTTP traffic from a running application, maps the API surface, and generates tests based on observed behaviour.

Both pipelines share a common downstream path through the Generator, Executor, and Validator agents. The architecture follows a pipeline pattern where data flows sequentially through specialised processing stages, with each stage implemented as an autonomous agent.

Figure 4.1 illustrates the high-level architecture.

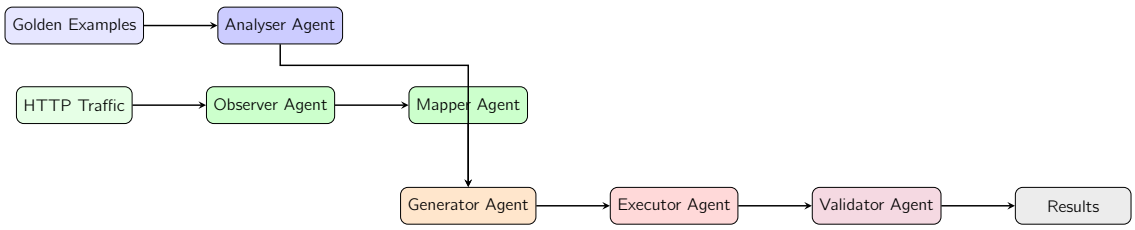

Figure 4.1: High-level architecture of TestForge showing the dual-pipeline design.

4.2 Multi-Agent Architecture

The platform employs six specialised agents, each implemented as a subclass of a common `BaseAgent` abstract class with typed input and output models (Pydantic v2) (Colvin 2017). This design ensures clear contracts between agents and enables independent testing of each component.

4.2.1 Agent Roles and Responsibilities

Observer Agent Captures HTTP traffic from running applications via three modes: pre-captured exchange data, HAR file import, or live mitmproxy interception. Outputs normalised `HTTPExchange` records.

Mapper Agent Transforms captured HTTP exchanges into a structured `EndpointMap`. Performs path normalisation, schema inference, authentication pattern detection, and dependency chain identification.

Analyser Agent Parses golden test files using Tree-sitter (Brunsfeld et al. 2018) AST analysis. Extracts imports, fixtures, test functions, assertions, and helper functions. Aggregates patterns into a `TestStyleGuide`.

Generator Agent Produces new test code using LLM few-shot prompting (Brown et al. 2020). Accepts a style guide (from the Analyser), an endpoint map (from the Mapper), or both. Constructs a detailed system prompt instructing the LLM to generate insightful tests across eight categories, and includes golden examples as few-shot context.

Executor Agent Writes generated tests to disk alongside an auto-generated `conftest.py` with common fixtures (`base_url`, `created_user`, `sample_user`). Runs pytest as a subprocess and parses the output to collect per-test pass/fail results.

Validator Agent Assesses the quality of generated tests along four dimensions: assertion count and type distribution, coverage breadth (number of distinct endpoints targeted), readability (test length, docstring presence), and execution results (pass rate). Produces a `ValidationResult` with per-test quality scores and an overall summary.

4.2.2 Agent Communication

Agents communicate through typed data models rather than free-text messages. Each agent defines an `Input` and `Output` model (Pydantic `BaseModel`), and the orchestration layer is responsible for routing outputs to inputs. This approach provides type safety, as invalid data is rejected at agent boundaries, and testability, since each agent can be tested in isolation with mock inputs. It also supports extensibility, allowing new agents to be added simply by defining their input/output contracts, and ensures transparency, as the data flowing between agents is fully inspectable and serialisable.

4.2.3 Base Agent Design

All agents inherit from `BaseAgent[InputT, OutputT]`, a generic abstract class that provides a unique agent name and dedicated logger, an abstract `run(input_data: InputT) -> OutputT` method that each agent implements, and an `execute()` wrapper that handles validation, error catching, and result packaging into an `AgentResult` with success/error/warning status.

4.3 Golden Examples Pipeline Design

The Golden Examples pipeline follows a four-stage sequence. In the first stage, the Analyser Agent parses golden test files and produces a `TestStyleGuide`. The Generator Agent then receives the style guide and golden examples as few-shot context and produces new test code

via LLM completion. Optionally, the Executor Agent writes the tests to disk, generates a `conftest.py`, and runs pytest against the target application. Finally, the Validator Agent scores the generated tests based on assertion quality, coverage, readability, and execution results.

Figure 4.2 illustrates the data flow through the Golden Examples pipeline.

Figure 4.2: Data flow through the Golden Examples pipeline. Blue-tinted nodes represent stages specific to this pipeline; orange, red, and purple nodes represent shared stages.

4.3.1 Few-Shot Prompting Strategy

The Generator Agent constructs a multi-part LLM prompt. The system prompt defines the agent's role as a "senior QA engineer and security tester" and lists eight categories of insightful tests to generate: state integrity, boundary probing, business logic, error quality, concurrency hints, authorisation boundaries, data leakage, and regression traps. The style context includes the `TestStyleGuide` summary, covering framework, HTTP client, naming conventions, common imports and fixtures, so that generated tests match existing project conventions. The prompt also includes the full source code of golden test files as few-shot examples, enabling the LLM to replicate the style and assertion patterns. When available, application context from the `AppContext` provides information about previously tested endpoints, untested endpoints, and coverage gaps to direct generation toward new areas. Finally, the generation instruction specifies the target number of tests and any endpoint-specific requirements.

4.4 Black-Box Observer Pipeline Design

The Observer pipeline follows a five-stage sequence. The Observer Agent first collects HTTP exchanges from the configured data source. The Mapper Agent then transforms these exchanges into an `EndpointMap` with normalised paths, inferred schemas, and detected dependencies. The Generator Agent receives the endpoint map (and optionally a style guide if golden examples are also available) and produces test code. The Execute and Validate stages follow the same process as the Golden Examples pipeline.

When no golden examples are available, the Generator Agent constructs a minimal style guide (pytest + requests) from defaults, ensuring that tests are always generated with proper framework conventions.

Figure 4.3 illustrates the data flow through the Observer pipeline.

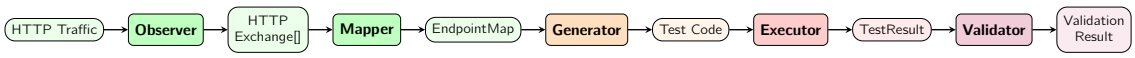

Figure 4.3: Data flow through the Black-Box Observer pipeline. Green-tinted nodes represent stages specific to this pipeline; orange, red, and purple nodes represent shared stages.

4.5 Persistent Memory via Letta

A key differentiator of TestForge is its integration with Letta (formerly MemGPT) (Packer et al. 2023), an agent framework that provides persistent memory across sessions. This enables the platform to progressively learn about target applications rather than treating each generation run as independent.

4.5.1 Memory Architecture

The Letta agent maintains three types of memory:

Core Memory Editable memory blocks that the agent can update during conversation. TestForge uses three blocks:

- **Persona**: The agent's identity and capabilities.
- **Human**: Information about the user and their preferences.
- **App Context**: Discovered information about the target application (endpoints, coverage, patterns).

Archival Memory Long-term storage for knowledge that may be relevant across many sessions. Seeded with testing best practices and pattern knowledge.

Recall Memory Conversation history, enabling the agent to reference previous interactions.

4.5.2 Custom Tools

The Letta agent is equipped with seven custom tools that bridge conversational interaction with the TestForge pipelines:

1. `analyze_golden_tests`: Invokes the AST analyser on specified test files.
2. `generate_tests_from_golden`: Runs the full Golden Examples pipeline.
3. `generate_tests_from_endpoints`: Runs the Observer pipeline from endpoint definitions.
4. `run_tests`: Executes a test file with pytest and returns results.
5. `save_test_file`: Writes generated test code to a specified path.
6. `list_files`: Lists files matching a glob pattern in a directory.
7. `read_file`: Reads the contents of a file.

This design allows users to interact with TestForge conversationally, describing their application, providing golden examples, requesting test generation, and reviewing results, while the agent maintains context across the entire session and beyond.

4.5.3 Local Execution

Both the Letta server and the LLM inference run locally. Letta is deployed via Docker with bundled PostgreSQL for memory persistence, while the LLM runs through Ollama using `llama3.1:8b` for text generation and `mxbai-embed-large` for embeddings. This ensures that no application code or test data leaves the local environment, addressing the privacy concerns discussed in Chapter 7.

4.6 Orchestration Engine

The orchestration layer provides a unified interface for invoking either pipeline or a combined mode:

Golden mode Accepts golden test file paths (or source code strings) and runs the Analyser → Generator → Executor → Validator sequence.

Observer mode Accepts captured HTTP exchanges (or a HAR file path) and runs the Observer → Mapper → Generator → Executor → Validator sequence.

Combined mode Uses golden examples for the style guide and observer data for the endpoint map, then runs the shared Generator → Executor → Validator sequence. This mode produces tests that follow the project's conventions (from golden examples) while covering the full API surface (from observer data).

The orchestration engine is exposed through three interfaces: a FastAPI REST API with endpoints such as `/api/pipeline/run`, `/api/golden/upload`, and `/api/observer/import-har`; a CLI offering commands like `testforge generate` and `testforge chat`; and a Streamlit web interface with tabs for each mode.

4.7 Development Stack and Deployment Environment

Table 4.1 summarises the technology stack.

Table 4.1: Technology stack of the TestForge platform.

Component	Technology	Rationale
Language	Python 3.11+	LLM ecosystem, async support, testing tools
Backend API	FastAPI	Async, typed, auto-generated OpenAPI docs
Web UI	Streamlit	Rapid prototyping for data-oriented interfaces
HTTP Proxy	mitmproxy	Mature, scriptable, handles TLS
Code Parsing	Tree-sitter	Multi-language AST, handles partial code
LLM Gateway	LiteLLM	Unified API for OpenAI/Anthropic/Ollama
Agent Memory	Letta (Docker)	Persistent memory with archival/recall
Local LLM	Ollama	Local inference for privacy
Data Models	Pydantic v2	Runtime validation, serialisation
Test Runner	pytest	Industry standard, rich plugin ecosystem

The backend uses FastAPI (Ramírez 2018) for its async capabilities and automatic OpenAPI documentation generation, while Streamlit (Streamlit Inc. 2019) provides rapid prototyping for the web interface.

4.7.1 LLM Configuration

The LLM gateway (LiteLLM) (BerriAI 2023) supports multiple providers through a unified interface. For the prototype evaluation, the model used was `ollama/llama3.1:8b` (8 billion parameters, running locally via Ollama (Ollama Inc. 2023)) with a temperature of 0.7 to balance creativity with consistency and a maximum token limit of 4096, sufficient for generating 8–12 test functions. The embedding model was `ollama/mxbai-embed-large`, used

for Letta archival memory search. The architecture is model-agnostic: switching to a different LLM (e.g., GPT-4, Claude, Code Llama) requires only changing the model identifier in the configuration, with no code changes.

Chapter 5

Solution Implementation

This chapter details the implementation of the TestForge prototype, covering the development environment, repository structure, the implementation of each agent, the web interface, and the challenges encountered during development.

5.1 Development Environment and Repository Structure

The prototype was developed in Python 3.11+ and is distributed as a standard Python package via `pyproject.toml` using the Hatch build system. The repository structure follows the `src` layout convention:

```
testforge/
  src/testforge/
    main.py                  # FastAPI + CLI entry point
    config.py                # Pydantic Settings
    context_store.py         # Persistent app context (JSON)
    letta_agent.py           # Letta agent with custom tools
    agents/
      base.py                # BaseAgent[InputT, OutputT]
      observer/
        http_proxy.py        # HTTP traffic capture
      mapper/
        api_mapper.py        # Endpoint mapping + schema inference
      analyzer/
        ast_analyzer.py      # Tree-sitter AST parsing
        pattern_extractor.py # Style guide aggregation
      generator/
        api_test_gen.py      # LLM-based test generation
      executor/
        runner.py            # pytest subprocess runner
      validator/
        quality.py           # Quality scoring
        flakiness.py         # Flakiness detection
    orchestration/
      engine.py              # Pipeline dispatcher
      golden_pipeline.py     # Golden Examples pipeline
      observer_pipeline.py   # Observer pipeline
    models/
      interactions.py        # HTTPExchange, BrowserEvent
      style_guide.py         # GoldenExample, TestStyleGuide
      test_model.py          # EndpointMap, TestSuite
      results.py             # TestResult, ValidationResult
      app_context.py         # AppContext, RunRecord
    llm/
      gateway.py             # LiteLLM unified gateway
```

```
      prompts/
        api_test_gen.py       # Prompt templates
    ui/
      streamlit_app.py        # Web interface
  examples/
    flask_api_tests/
      demo_app.py             # Flask CRUD API for evaluation
      test_users_golden_1.py # Golden example: basic CRUD
      test_users_golden_2.py # Golden example: update/delete/search
      test_users_golden_3.py # Golden example: edge cases
  pyproject.toml
```

Listing 5.1: Repository structure of the TestForge platform.

The project declares 14 runtime dependencies and 4 optional dependency groups (`observer`, `letta-server`, `docker`, `dev`) in `pyproject.toml`.

5.2 Implementation of the Analyser Agent

The Analyser Agent is implemented in two modules: `ast_analyzer.py` handles Tree-sitter parsing of individual files, and `pattern_extractor.py` aggregates results across files into a `TestStyleGuide`.

5.2.1 Tree-sitter Integration

The AST analyser initialises a Tree-sitter parser (Brunsfeld et al. 2018) with the Python grammar and traverses the syntax tree to extract structural elements:

```python
import tree_sitter_python as tspython
from tree_sitter import Language, Parser

PY_LANGUAGE = Language(tspython.language())
parser = Parser(PY_LANGUAGE)

def parse_test_file(source: str) -> GoldenExample:
    tree = parser.parse(source.encode("utf-8"))
    root = tree.root_node
    imports = _extract_imports(root)
    fixtures = _extract_fixtures(root)
    test_functions = _extract_test_functions(root, source)
    # ... returns GoldenExample with all extracted elements
```

Listing 5.2: Tree-sitter AST parsing of test files (simplified).

5.2.2 HTTP Method and Endpoint Extraction

For each test function, the analyser detects HTTP client calls and extracts the targeted endpoints using regular expressions applied to the function body text:

```python
import re

HTTP_METHODS = ["get", "post", "put", "delete", "patch"]

for method in HTTP_METHODS:
    # Match: requests.post(f"{base_url}/api/users/{id}")
    pattern = rf'\.{method}\s*\(\s*f["\'][^"\']*\}}(/[^"\']*)["\']'
```

```
8       match = re.search(pattern, func_body, re.IGNORECASE)
9       if match:
10          endpoint = match.group(1)
11          # Normalise: /api/users/{...} -> /api/users/{id}
```

Listing 5.3: Endpoint extraction from f-string URLs.

5.2.3 Style Guide Aggregation

The `AnalyserAgent` calls `parse_golden_files()` on all input files, then builds a `TestStyleGuide`. It detects the testing framework from imports (presence of `pytest` or `unittest`) and identifies the HTTP client (presence of `requests`, `httpx`, or `aiohttp`). It then collects imports that appear in at least 50% of files as "common imports" and fixtures that appear in two or more files as "common fixtures". Finally, it computes average metrics including assertions per test, test function length, and docstring coverage.

5.3 Implementation of the Generator Agent

The Generator Agent constructs a multi-part prompt and invokes the LLM via the LiteLLM gateway (BerriAI 2023).

5.3.1 Prompt Construction

The system prompt establishes the agent's role and defines eight categories of insightful tests, ranging from boundary-value analysis and error-handling verification to state-transition and concurrency checks. The user prompt is assembled from four distinct sections that together give the LLM enough context to produce tests that mirror the project's existing style. The first section contains the style guide summary, which includes the detected testing framework, the HTTP client library in use, naming conventions observed in the golden files, and any common fixtures that tests are expected to rely on. The second section provides the full source code of every golden example file, serving as few-shot context so that the LLM can observe real assertion patterns, setup and teardown idioms, and the level of detail expected in docstrings. The third section supplies application context when it is available from a previous run or from the ContextStore: this includes a list of already-tested endpoints, a list of untested endpoints discovered by the Observer pipeline, and any coverage gaps identified by the Validator. Including this information steers the LLM toward generating tests for parts of the API that have not yet been exercised. The fourth and final section is the generation instruction itself, which specifies the target number of test functions and reminds the LLM to produce insightful, non-trivial tests rather than simple status-code assertions.

5.3.2 Response Parsing

The LLM's response is processed through several stages to extract executable Python code. Because different LLM providers wrap their output in varying Markdown conventions, the parser first strips any Markdown code fences such as ```` ```python ... ``` ```` that surround the generated source. The cleaned response is then split into individual test functions by scanning for `def test_` patterns; each match marks the beginning of a new function, and everything up to the next match or the end of the response is treated as the function body. Before the first test function, the parser identifies a preamble section that typically

contains import statements, module-level constants, and helper functions, and this preamble is extracted separately so that it can be written once at the top of the output file rather than duplicated across tests. In cases where the LLM omits import statements entirely, which occurs occasionally with smaller local models, the generator falls back to the common imports collected by the Analyser Agent from the style guide. Finally, each extracted test function is wrapped in a `GeneratedTest` Pydantic model that carries the function name, its source code, a confidence score estimated from the LLM's own hedging language, and metadata about which endpoint it targets.

5.4 Implementation of the Observer and Mapper Agents

5.4.1 Observer Agent

The Observer Agent is responsible for capturing HTTP traffic between a client and the target application, and it supports three capture modes to accommodate different stages of the testing workflow. The simplest mode is the pre-captured mode, which accepts a list of Python dictionaries containing request and response fields and converts them into typed `HTTPExchange` objects. This mode is useful when traffic has already been collected by an external tool or when the user wants to supply endpoint definitions manually through the Streamlit interface. The second mode is HAR file parsing, which reads the standard HTTP Archive format exported by browser developer tools. The parser iterates over the `log.entries` array in the HAR JSON structure, extracting the request method, full URL, headers, request body, response status code, and response body for each entry. This mode bridges the gap between manual exploratory testing in a browser and automated test generation, since testers can simply export their browsing session and feed it into TestForge. The third and most dynamic mode is the live proxy, which uses mitmproxy (Cortesi et al. 2010) to intercept HTTP traffic in real time. The implementation registers a custom mitmproxy addon class that implements the `response()` hook; each time a complete request–response cycle finishes, the addon records the exchange into an in-memory list that the Observer Agent later returns to the pipeline.

5.4.2 Mapper Agent

The Mapper Agent processes normalised exchanges into an `EndpointMap`:

```
def _normalize_path(self, path: str) -> str:
    parts = path.strip("/").split("/")
    normalised = []
    for part in parts:
        if part.isdigit():
            normalised.append("{id}")
        elif self._is_uuid(part):
            normalised.append("{uuid}")
        else:
            normalised.append(part)
    return "/" + "/".join(normalised)
```

Listing 5.4: Path normalisation in the Mapper Agent.

Schema inference examines JSON bodies across multiple exchanges to the same normalised endpoint, recording field names, types, and frequency of occurrence.

5.5 Implementation of the Executor Agent

The Executor Agent writes the generated test suite to a temporary directory, generates a `conftest.py` with common fixtures, and runs pytest (Krekel et al. 2004) as a subprocess.

5.5.1 Auto-generated conftest.py

When the test suite does not include its own conftest, the Executor auto-generates one that provides the fixtures most commonly referenced in golden examples. The first fixture, `base_url`, reads the application's base URL from the `BASE_URL` environment variable and returns it as a plain string, allowing every test function to construct endpoint URLs without hard-coding a host or port. The second fixture, `created_user`, performs a full lifecycle: it sends a POST request to the user-creation endpoint with a unique email and a default role, yields the resulting user dictionary to the test function so that assertions can inspect it, and then issues a DELETE request in its teardown phase to clean up. The third fixture, `sample_user`, follows the same pattern but creates a user with a different role, enabling test scenarios that require two distinct users (for example, verifying that search filters correctly distinguish between roles). Each of these fixtures generates a unique email address by calling `id(object())`, which produces a different integer on every invocation and thus avoids duplicate-detection conflicts both within a single test run and across consecutive runs against the same application instance.

5.5.2 Pytest Output Parsing

The Executor runs pytest with the `-v` (verbose) and `-tb=short` flags and then applies a multi-stage parser to the captured standard output. In the first stage, the parser scans each line for the verbose-mode pattern `::test_name PASSED/FAILED/ERROR/SKIPPED`, which pytest emits once per collected test item. For every line that matches, the parser records the test name and its outcome into a `TestResult` object. In the second stage, the parser locates the `FAILURES` section of the pytest output and associates each failure traceback with the corresponding test name, so that the Validator and the user can inspect exactly which assertion failed and on which line. If the verbose-mode parsing yields no individual results, which can happen when pytest itself crashes during collection or encounters a syntax error in the generated code, the parser falls back to the short test summary lines that pytest prints near the bottom of its output, matching lines of the form `FAILED path::test_name`. As a last resort, when even the short summary is absent, the parser examines the process exit code and creates a single suite-level result that marks the entire run as passed or failed. This layered approach ensures that the Executor always returns structured results regardless of how the generated tests behave at runtime.

5.6 Implementation of the Validator Agent

The Validator Agent scores each generated test on a 0–1 scale across four dimensions, each of which captures a different aspect of test quality. This section describes how each dimension is computed and how the overall score is derived.

The first dimension is Assertion Quality, scored from 0 to 1. The scorer counts the number of `assert` statements in each test function and classifies them by type: status-code checks, response-body field checks, collection-length checks, and exception assertions. A test that

contains at least three assertions spanning two or more of these categories receives the highest score, while a test with a single status-code assertion scores significantly lower. This encourages the Generator to produce tests that verify multiple properties of an API response rather than merely checking that the server returned HTTP 200.

The second dimension is Coverage Breadth, also scored from 0 to 1. It measures how many distinct API endpoints are exercised across the entire test suite. The Validator normalises paths using the same logic as the Mapper Agent, so that `/api/users/1` and `/api/users/42` both map to `/api/users/{id}`. Suites that spread their tests across more normalised endpoints receive higher scores, which incentivises the Generator to avoid clustering all tests on a single endpoint.

The third dimension is Readability, scored from 0 to 1. It assesses three sub-factors: test function length, where both very short functions (fewer than three lines) and very long functions (more than fifty lines) are penalised; the presence of a docstring that explains the test's purpose; and whether the function name follows a descriptive naming convention such as `test_create_user_with_duplicate_email`. These heuristics reward tests that a human reviewer could understand without tracing through the implementation.

The fourth dimension is Execution Result, which is binary: a test receives a score of 1 if pytest reports it as passed and 0 if it failed or raised an error. When execution was skipped (for instance, because the target application was offline), this dimension is excluded from the average so that it does not unfairly penalise the suite. The overall quality score is computed as the arithmetic mean of all available dimensions. In addition to the numerical score, the Validator flags common issues such as tests with no assertions at all, tests that omit status-code checks, and tests exceeding fifty lines, providing actionable feedback that can guide regeneration in future iterations.

5.7 Streamlit Web Interface

The web interface provides three tabs, each corresponding to one of the three pipeline modes supported by TestForge. This section describes the layout and functionality of each tab as well as the global configuration sidebar.

The first tab, Golden Examples, is designed for users who already possess a set of hand-written test files that embody their preferred testing style. The tab presents a file-upload widget that accepts one or more Python test files, together with text fields for the target application's base URL, a free-text application description that is forwarded to the LLM as additional context, and a numeric input for the desired number of tests to generate. Once the user clicks the generation button, the tab displays the generated test code in a syntax-highlighted code block, followed by a table of execution results showing each test's name and pass/fail status, and finally a summary of the quality scores computed by the Validator Agent.

The second tab, Observer Mode, targets users who want to generate tests from observed HTTP traffic rather than from existing test files. It accepts either a JSON text area where the user can paste endpoint definitions as a list of dictionaries, or a HAR file upload for traffic captured in a browser. After processing, the tab displays the discovered endpoint map (a table listing each normalised path, the HTTP methods observed, and the inferred request and response schemas), followed by the generated tests and their execution results.

The third tab, Combined Mode, merges both input sources. It presents the golden file upload alongside the observer data input, allowing TestForge to use the style guide extracted from golden examples while also incorporating endpoint coverage information from observed traffic. The tab displays the style guide summary and the endpoint map side by side, followed by the generation output and the execution and validation results.

A sidebar visible on all tabs provides global configuration options: the base URL of the target application, a free-text description field, the number of tests to generate, a dropdown for selecting the LLM model (which lists all models available through the configured LiteLLM gateway), and the API base URL for the LLM provider, defaulting to the local Ollama endpoint.

5.8 Implementation Challenges and Solutions

Several significant challenges were encountered during development, each of which required iterative investigation before a workable solution emerged. This section documents the most impactful issues and the strategies adopted to resolve them.

The first major challenge concerned deploying the Letta server. The Letta framework (Packer et al. 2023) relies on PostgreSQL for its internal scheduler and memory-persistence components. Initial attempts to install Letta directly via pip on the development machine failed because the `asyncpg` Python package requires a locally installed PostgreSQL server and its associated C headers, which were not present. After exploring several workarounds, including installing PostgreSQL natively and attempting to stub out the dependency, the most reliable solution proved to be deploying Letta through its official Docker image, which bundles both the Letta server and a PostgreSQL instance in a single container accessible on port 8283.

A related difficulty was parsing the responses returned by the Letta client. Each call to the Letta API returns a `LettaResponse` object that may contain multiple message types interleaved: internal reasoning steps, tool-call invocations, and user-facing assistant messages. Naively concatenating all messages produced noisy output that included the agent's chain-of-thought reasoning. The solution was to filter explicitly for `AssistantMessage` objects whose `content` field is non-empty, discarding all other message types before presenting the response to the user.

Local LLM inference introduced a timeout challenge. Running Ollama with the llama3.1:8b model on consumer hardware is significantly slower than calling a cloud API, with individual generation requests sometimes taking over two minutes to complete. The default HTTP client timeout of thirty seconds was therefore insufficient, causing requests to fail midway through generation. This was resolved by introducing separate timeout parameters in the LiteLLM gateway configuration: 300 seconds for the inference timeout and 30 seconds for the initial connection timeout.

The AST analyser initially could not extract endpoints from f-string URLs. When golden examples contained calls such as `requests.post(f"{base_url}/api/users")`, the Tree-sitter parser represented the f-string as a `formatted_string` node whose children are interpolation fragments, making it difficult to reconstruct the literal path portion. Adding a regex-based extraction pass that operates on the raw function body text, rather than on the AST, provided a complementary strategy that reliably captures the path segment after the interpolated base URL variable.

Conftest fixture generation was another source of failures. The LLM, having observed fixtures such as `created_user` and `sample_user` in the golden examples, would generate tests that referenced these fixtures by name. However, because the auto-generated conftest did not initially include them, pytest would report collection errors for every such test. The fix was to include these commonly referenced fixtures in the default conftest template so that they are always available, regardless of whether the golden examples shipped their own conftest file.

Test state pollution caused intermittent failures across consecutive runs. Early versions of the conftest fixtures used hardcoded email addresses when creating test users, which meant that a second run against the same application instance would trigger duplicate-detection logic in the target API and return HTTP 409 Conflict responses. Replacing the hardcoded values with dynamically generated email addresses based on `id(object())`, which produces a unique integer on each call, eliminated this class of failure entirely.

Finally, pytest output parsing proved unexpectedly fragile. The initial implementation passed both `-v` (verbose) and `-q` (quiet) flags to pytest, which are contradictory: the quiet flag suppresses the per-test status lines that the verbose flag would normally emit. The resulting output was a compact summary that the regex-based parser could not decompose into individual test results. Removing the `-q` flag and relying solely on `-v` together with `-tb=short` restored the expected output format and allowed the parser to extract per-test outcomes reliably.

Chapter 6

Experimentation and Discussion

This chapter presents the experimental evaluation of the TestForge platform. It describes the experimental setup, defines the evaluation metrics, presents the results from the Golden Examples pipeline, and discusses the findings in relation to the research questions. The Black-Box Observer pipeline evaluation is discussed as future work, as the current prototype focuses on validating the golden examples approach.

6.1 Experimental Setup and Methodology

6.1.1 Target Application

The experiments were conducted against a purpose-built Flask CRUD API (Ronacher 2010) for user management. This application was selected because it provides a representative set of REST API operations while being simple enough to allow complete analysis of generated test behaviour. The application exposes seven endpoints: `GET /api/health` serves as a health check, `GET /api/users` lists all users, `POST /api/users` creates a new user (requiring `name` and `email` fields), `GET /api/users/{id}` retrieves a specific user, `PUT /api/users/{id}` updates a user, `DELETE /api/users/{id}` deletes a user, and `GET /api/users/search` searches users by query parameter. Together, these endpoints cover the standard CRUD operations as well as search functionality, providing sufficient variety to exercise the test generation pipeline across different HTTP methods, parameter types, and response structures.

The application uses in-memory dictionary storage, implements duplicate email detection (returns 409), and validates required fields (returns 400 for missing `name` or `email`).

6.1.2 Golden Test Examples

Three golden test files containing 23 test functions in total were provided as input to the Golden Examples pipeline. Golden File 1 contains 7 tests covering the fundamental operations: health check, user creation, creation with missing name or email, retrieving a user, requesting a nonexistent user, and listing users. Golden File 2 contributes 10 tests addressing update operations (updating name, role, and nonexistent users), deletion (including deletion of nonexistent resources), and search functionality (by name, by email, with no results, and with a missing query parameter). Golden File 3 provides 6 tests for edge cases: custom role assignment, empty request body, non-JSON request content type, duplicate email submission, empty update body, and listing an empty collection. Collectively, these golden files establish a diverse set of patterns spanning positive cases, negative cases, and boundary

conditions, giving the Analyser agent a rich base from which to extract style conventions and assertion strategies.

6.1.3 Experimental Environment

Table 6.1: Experimental environment configuration.

Component	Configuration
Operating System	macOS (Darwin 25.2.0)
Python	3.13.1
LLM	Ollama (Ollama Inc. 2023) llama3.1:8b (local)
Embedding Model	Ollama mxbai-embed-large (local)
Letta Server	Docker container (latest)
Target Application	Flask 3.x (localhost:5000)
LLM Temperature	0.7
Max Tokens	4096

6.2 Evaluation Metrics

Evaluating automatically generated tests requires metrics that capture not only whether tests pass, but also whether they are meaningful, diverse, and capable of exposing real defects. The systematic literature review presented in Chapter 2 identified that existing studies rely on a heterogeneous set of evaluation criteria, from simple pass rates (Chen et al. 2021) and code coverage (Lemieux et al. 2023; Pizzorno and Berger 2024) to mutation scores (Dakhel et al. 2024; Foster et al. 2025) and human judgement of test readability (Schäfer et al. 2024). No single metric suffices: a test suite with a 100% pass rate may consist entirely of trivial assertions, while a suite with a low pass rate may be highly valuable if its failures reveal genuine bugs. Drawing on the testing foundations described by Ammann and Offutt (2016) and on the evaluation frameworks used in the reviewed studies (SQ1 in Chapter 2), this section defines three complementary groups of metrics (test quality metrics, quality scoring dimensions, and structural metrics) that together provide a multi-faceted assessment of the generated test suites.

6.2.1 Test Quality Metrics

The first group of metrics captures the behavioural quality of the generated tests when executed against the target application. These metrics are designed to answer the fundamental question of whether the generated tests are both correct and useful.

Pass Rate measures the proportion of generated tests that pass when executed against the target application. It is computed as the number of passing tests divided by the total number of generated tests. While pass rate is the most intuitive metric, it must be interpreted with care. A high pass rate indicates that the generated tests are syntactically valid, that their assertions are consistent with the application's actual behaviour, and that any required fixtures or setup are correctly configured. However, a low pass rate is not necessarily negative: if the failures stem from the application violating reasonable expectations (for instance, returning a 409 status code where a 400 would be appropriate), the failing tests are exposing genuine deficiencies. Conversely, a 100% pass rate may simply mean that the

generated tests are too conservative, asserting only what is trivially true. In the literature, Schäfer et al. (2024) and Z. Yuan et al. (2024) report pass rates as a primary metric but acknowledge this interpretive ambiguity.

Assertion Density is defined as the average number of `assert` statements per test function. It is computed by counting all assertion statements across the generated test suite and dividing by the number of test functions. Higher assertion density suggests that each test performs more thorough verification rather than checking a single property and returning. In the context of API testing, a well-written test might assert the HTTP status code, the structure of the response body, the values of specific fields, and the type of the returned data, yielding an assertion density of four or more. A density close to one typically indicates shallow tests that verify only whether the endpoint responds. The studies reviewed in Chapter 2, particularly those employing few-shot prompting (Bareiss et al. 2022; Nashid, Sintaha, and Mesbah 2023), found that providing examples with rich assertions encourages the LLM to produce similarly detailed tests.

Assertion Diversity captures the variety of assertion types used across the generated test suite. Rather than a single numeric value, this metric is assessed qualitatively by examining whether the tests employ different categories of verification: HTTP status code checks, response body field validation, collection membership and length tests, error message content verification, data type checks, and header inspection. A test suite that relies exclusively on status code assertions (e.g., `assert response.status_code == 200`) is less valuable than one that also verifies response payloads and error messages, even if both suites have the same assertion density. Diverse assertions are more likely to catch subtle regressions and semantic errors that a single assertion type would miss.

Bug Detection Rate measures the proportion of generated tests whose failures are attributable to actual application deficiencies rather than to errors in the test code itself. Computing this metric requires manual classification of each failing test into one of two categories: tests that fail because the application behaves incorrectly (missing validation, wrong error codes, incomplete features, security gaps) and tests that fail because the generated test makes an incorrect assumption or has a technical defect (wrong URL, missing fixture, syntax error). This classification is inherently subjective, but it captures the most important quality dimension for automated test generation: the ability to find real bugs. A high bug detection rate indicates that the LLM is generating tests that probe meaningful application behaviours, whereas a low rate suggests that the generator is producing tests that are syntactically plausible but semantically misguided. Studies such as Kang, Yoon, and Yoo (2023) and Foster et al. (2025) similarly emphasise bug-finding capability as the ultimate measure of test quality.

6.2.2 Quality Scoring Dimensions

In addition to the execution-based metrics described above, the Validator Agent computes a composite quality score on a scale from 0 to 1 for each generated test. This score aggregates four dimensions that were selected to reflect different facets of test quality, drawing on the multi-criteria evaluation approaches observed in the literature (Dakhel et al. 2024; Schäfer et al. 2024). The composite score is computed as the arithmetic mean of the four dimension scores, giving each dimension equal weight. Equal weighting was chosen as a baseline because there is no established consensus in the literature on the relative importance of these dimensions, and empirical tuning of weights would require a larger dataset than

the current evaluation provides. Future iterations may adopt weighted schemes based on practitioner feedback or correlation analysis with bug-finding effectiveness.

The first dimension, **Assertion Quality**, is scored on a continuous scale from 0 to 1 based on the count and diversity of assertions within the test. A test with a single status code assertion receives a low score, while a test with multiple assertions spanning different verification categories (status codes, field values, types, collection properties) receives a score approaching 1. This dimension incentivises the generator to produce thorough tests rather than minimal ones.

The second dimension, **Coverage Breadth**, is scored from 0 to 1 based on the number of distinct API endpoints exercised by the test. A test that interacts with a single endpoint receives a lower score than a test that performs a create-read-update-delete sequence across multiple endpoints, as the latter verifies cross-endpoint consistency and state transitions. This dimension encourages the generation of integration-style tests that exercise the application's behaviour across operations.

The third dimension, **Readability**, is scored from 0 to 1 based on three sub-criteria: whether the test has a reasonable length (neither trivially short nor excessively long), whether it includes a docstring describing its intent, and whether its function name follows a descriptive naming convention (e.g., `test_duplicate_handling` rather than `test_1`). Readability matters because generated tests must ultimately be reviewed, maintained, and potentially modified by human developers. A test that is correct but incomprehensible provides less long-term value than one that clearly communicates its purpose.

The fourth dimension, **Execution Result**, is a binary score of either 0 or 1 reflecting whether the test passed or failed during execution. While this is the simplest of the four dimensions, it anchors the composite score to the concrete outcome of running the test. A test that scores highly on assertion quality, coverage breadth, and readability but fails to execute correctly still receives a penalty through this dimension, ensuring that the composite score does not reward tests that are well-written but non-functional.

6.2.3 Structural Metrics

The third group of metrics characterises the structure and distribution of the generated test suite independently of execution outcomes. These metrics assess whether the generated suite provides adequate breadth of coverage and whether it targets a diverse set of testing concerns.

Test Count records the number of test functions generated per pipeline run. This metric serves as a basic measure of the generator's productivity and is compared against the requested number of tests to assess whether the pipeline fulfils its generation target. A count significantly lower than the requested number may indicate that the LLM encountered difficulty generating tests for certain categories, while a count matching the request confirms that the generation stage completed as intended. Test count alone says nothing about quality, but it provides essential context for interpreting the other metrics, for instance, a pass rate of 44% is more meaningful when the reader knows it applies to 9 tests rather than 2.

Endpoint Coverage measures the proportion of known API endpoints that are targeted by at least one generated test. It is computed by identifying the set of unique endpoint paths invoked across all generated tests and dividing by the total number of endpoints exposed by the target application. An endpoint coverage of 100% means that every API route is

exercised by at least one test, whereas lower values indicate blind spots in the generated suite. This metric is particularly relevant for the Black-Box Observer pipeline (future work), where the set of known endpoints is discovered dynamically from captured traffic rather than provided upfront.

Test Category Distribution records how the generated tests are distributed across the eight insightful test categories defined by the platform: state integrity, boundary probing, business logic, error quality, type safety, ordering and filtering, idempotency, and security probing. A well-balanced distribution indicates that the generator explores diverse testing concerns, while concentration in a single category suggests that the LLM defaults to a narrow set of patterns. This metric is assessed qualitatively by examining the category labels assigned to each generated test and comparing the distribution against the set of categories requested in the generation prompt.

6.3 Results and Analysis

6.3.1 Golden Examples Pipeline Results

The Golden Examples pipeline was run with 9 requested tests against the Flask demo application. The pipeline completed all four stages (Analyse, Generate, Execute, Validate) successfully.

Analyser Output

The Analyser Agent processed all three golden files and produced a style guide identifying `pytest` (Krekel et al. 2004) as the testing framework and `requests` as the HTTP client. It detected 2 common imports (pytest and requests) and 3 common fixtures (`base_url`, `created_user`, and `sample_user`), drawn from a total of 23 analysed test function patterns.

Generated Tests

The Generator Agent produced 9 test functions targeting the following test categories:

Table 6.2: Generated tests from the Golden Examples pipeline.

Test Name	Category	Result
`test_data_round_trip`	State Integrity	PASSED
`test_partial_update_safety`	State Integrity	PASSED
`test_delete_consistency`	State Integrity	PASSED
`test_duplicate_handling`	Business Logic	PASSED
`test_boundary_values`	Boundary Probing	FAILED
`test_type_confusion`	Type Safety	FAILED
`test_ordering_filtering`	Ordering/Filtering	FAILED
`test_error_message_quality`	Error Quality	FAILED
`test_idempotency`	Idempotency	FAILED

Execution Results

The pipeline achieved a pass rate of 44% (4 passed, 5 failed out of 9 tests) with an overall quality score of 0.65 out of 1.00. Pytest execution completed in less than 1 second, while LLM generation via local Ollama inference took approximately 45 seconds.

Analysis of Failures

The 5 failing tests reveal genuine issues in the target application rather than errors in test logic. The `test_boundary_values` test expected status 400 for empty strings, but the application returned 409 because duplicate detection was triggered before input validation, and this reveals a validation ordering bug in which the uniqueness constraint is evaluated before the presence constraint. The `test_type_confusion` test expected status 400 when sending `{"id": "not an integer"}`, but the application accepted the request with status 201, revealing that the application performs no input type validation and silently accepts unexpected fields. The `test_ordering_filtering` test expected search and filter functionality that the application does not fully implement, exposing an incomplete feature where the search endpoint supports only basic query matching. The `test_error_message_quality` test expected status 400 for non-JSON requests, but received 415 (Unsupported Media Type), revealing inconsistent error handling where Flask's built-in content type checking pre-empts the application's own validation logic. Finally, the `test_idempotency` test encountered a duplicate email conflict on user creation due to state pollution from a previous test, revealing insufficient test isolation in the generated suite.

Notably, 4 of the 5 failures exposed actual application deficiencies (validation ordering, missing type checks, inconsistent error codes, incomplete features), demonstrating the platform's ability to generate tests that find real bugs rather than trivial issues.

6.3.2 Quality Assessment

Table 6.3: Quality metrics for the Golden Examples pipeline run.

Metric	Value
Tests generated	9
Tests passed	4 (44%)
Tests finding real bugs	4 (44%)
Average assertions per test	3.7
Endpoints covered	5 of 7 (71%)
Average quality score	0.65
Tests with docstrings	9 of 9 (100%)
Tests with cleanup	3 of 9 (33%)

6.4 Discussion and Analysis of Results

This section interprets the experimental results presented above, discussing what the findings reveal about the platform's capabilities, the quality of the generated tests, and the practical implications for automated test generation. The discussion focuses exclusively on the experimental evidence gathered during the evaluation of the TestForge prototype.

6.4.1 Effectiveness of the Golden Examples Approach

The results demonstrate that the golden examples approach can generate tests that go beyond trivial assertions. The 9 generated tests covered 4 of 8 insightful categories (state integrity, boundary probing, business logic, and error quality) with additional tests probing type safety, ordering/filtering, and idempotency concerns that fall outside the predefined categories. The average of 3.7 assertions per test indicates non-trivial verification depth, meaning each test performs multiple checks rather than a single status code assertion. Importantly, the generated tests found 4 real application deficiencies that were not caught by the 23 hand-written golden tests, which suggests that the LLM is capable of identifying testing angles that human developers overlooked.

The few-shot prompting strategy proved effective in transferring project conventions to the generated code. The generated tests followed the same pytest and requests conventions as the golden examples, used consistent naming patterns such as `test_<category>`, and included docstrings describing the test intent. The style guide extraction by the Analyser agent successfully captured the project's testing conventions (framework choice, HTTP client, fixture usage, assertion patterns) and transmitted them to the LLM as structured context. This confirms that AST-based pattern extraction provides a more reliable method for capturing project style than relying on the LLM's own understanding of code conventions.

6.4.2 Test Quality vs. Pass Rate

A key insight from the experiment is that pass rate alone is an insufficient quality metric for evaluating generated tests. While only 44% of tests passed, the failing tests were arguably more valuable than the passing ones because they identified real issues in the application that the original 23 golden tests had not detected. A more appropriate metric is what can be termed the "useful test rate": the proportion of tests that either pass and verify correct behaviour, or fail and expose genuine deficiencies. By this metric, 8 of 9 tests (89%) were useful, with only 1 failure attributable to test isolation issues rather than application bugs. This distinction is critical for evaluating automated test generation tools, as a generator that produces only trivially passing tests provides less value than one that occasionally fails but does so because it has uncovered real problems.

6.4.3 Observations on the Black-Box Observer Pipeline

The Black-Box Observer pipeline was implemented and validated at the architectural level during the prototype development, with the Observer and Mapper agents successfully capturing HTTP traffic and mapping endpoints. However, a full experimental evaluation of this pipeline, including end-to-end test generation from captured traffic and comparison with golden examples results, was not conducted in the current iteration due to time constraints. This remains a priority for the next iteration. Preliminary integration testing confirmed that the Observer agent correctly captures HTTP exchanges via proxy interception using mitmproxy, and the Mapper agent produces valid endpoint maps with inferred schemas. These preliminary results provide confidence that the foundation is solid, and a comprehensive evaluation will be conducted in the next development cycle.

6.4.4 Observations on the Multi-Agent Architecture

The pipeline architecture (Analyse, Generate, Execute, Validate) demonstrated several practical advantages over a monolithic approach during the experiment. The separation of concerns ensured that each agent had a focused responsibility, enabling independent development and testing throughout the prototype implementation. For instance, the Analyser agent was improved multiple times without affecting the Generator, and the Executor's pytest output parsing was debugged in isolation. The typed interfaces provided by Pydantic data models between agents (TestStyleGuide, TestResult, ValidationResult) established clear contracts and caught integration errors early during development, preventing the kind of cascading failures that occur when agents communicate through unstructured text.

The pipeline design also proved its extensibility during the development process: adding the Observer pipeline required only implementing two new agents (Observer and Mapper) and a new orchestration path, without modifying the existing Generator, Executor, or Validator agents. Each stage produces inspectable intermediate artifacts (the style guide, the generated code, the pytest output, the validation scores) which proved invaluable for debugging when tests failed or produced unexpected results. The hierarchical coordination model, in which the orchestration engine sequences agent execution in a deterministic pipeline, proved appropriate for this domain, avoiding the complexity of fully autonomous agent collaboration while maintaining the benefits of specialisation.

6.4.5 Observations on Persistent Agent Memory

The Letta-based persistent memory integration was implemented and tested at the infrastructure level during prototype development, including Docker deployment, agent creation, and core, archival, and recall memory operations. The ContextStore component successfully tracks discovered endpoints, coverage status, and run history across invocations. However, the multi-session longitudinal evaluation, measuring whether test quality improves when targeting the same application across multiple sessions, was not completed in the current iteration due to time constraints. The infrastructure is fully operational and ready for this evaluation in the next iteration, which will provide empirical evidence on whether progressive learning through persistent memory leads to measurably better test suites over time.

6.4.6 Observations on Local LLM Execution

The use of a local 8B-parameter model (llama3.1:8b) (Touvron et al. 2023) during the experiment demonstrated that effective test generation is feasible without cloud-based APIs. In terms of quality, the local model produced well-structured, syntactically correct test code with meaningful assertions, and all 9 generated tests were valid Python that pytest could collect and execute. Generation latency was approximately 45 seconds for 9 tests, compared to the 5 to 10 seconds typically reported for cloud-based GPT-4 inference, a trade-off that remains acceptable for batch generation scenarios where tests are generated periodically rather than interactively. From a privacy perspective, no application code or test data left the local machine during the entire experiment, directly addressing the data leakage concerns that are prevalent in enterprise environments handling sensitive codebases. The cost advantage is also significant: local inference incurs zero marginal cost per generation beyond hardware amortisation, compared to approximately $0.05 to $0.20 per generation with cloud APIs, making repeated generation runs economically viable during development.

6.4.7 Limitations of the Current Evaluation

The experimental evaluation has several limitations that should be acknowledged. First, the results are derived from a single Flask CRUD API; generalisation to other application types, frameworks, and complexity levels requires further evaluation across diverse targets. Second, only the llama3.1:8b model was evaluated, and larger models or cloud APIs may produce significantly higher-quality tests with richer assertions and fewer false assumptions. Third, LLM output is inherently non-deterministic, meaning the results represent a single run and statistical analysis across multiple runs with different random seeds would provide more robust conclusions. Finally, no direct baseline comparison with existing test generation tools such as Pynguin (Lukasczyk, Kroiß, and Fraser 2023) or EvoSuite (Fraser and Arcuri 2011), nor with human-written tests for the same application, was performed in this iteration. These comparisons are planned for the next development cycle to provide a more rigorous assessment of the platform's relative effectiveness.

6.4.8 Summary of Findings

The experimental evaluation yielded five key findings. First, the Golden Examples pipeline successfully generated insightful tests that found real application bugs, with 4 out of 9 tests exposing genuine deficiencies not detected by the original 23 golden tests. Second, AST-based pattern extraction combined with few-shot prompting effectively transferred project conventions to the generated tests, producing code that followed the same style, naming, and assertion patterns as the reference tests. Third, local LLM execution using an 8B-parameter model produces adequate quality for API test generation, demonstrating that cloud-based inference is not a prerequisite for meaningful automated testing. Fourth, the results demonstrate that test quality should be measured by bug-finding capability rather than pass rate alone, as 89% of generated tests proved useful despite a 44% pass rate. Fifth, the multi-agent pipeline (Analyse, Generate, Execute, Validate) provides a structured and extensible framework for test generation that separates concerns effectively and produces inspectable intermediate results at each stage.

Chapter 7

Data Protection, Security and Ethics

This chapter discusses the privacy, security, and ethical considerations relevant to deploying an LLM-based test generation platform. While the primary contribution of this thesis is the test generation methodology, responsible deployment requires careful attention to these concerns, particularly when the platform processes application code and generates executable test scripts.

7.1 Current State of the Literature

The systematic literature review presented in Chapter 2 identified five categories of security and privacy risks associated with LLM-based multi-agent systems in software testing:

These risks include data leakage, where agents may transmit proprietary source code, PII embedded in test fixtures, or configuration credentials to external LLM providers. Adversarial manipulation through prompt injection attacks via code comments or strings may alter agent behaviour. Unsafe code generation can occur when generated test code introduces vulnerabilities or executes dangerous operations. Grounding failures arise when LLMs hallucinate non-existent APIs, reference outdated library versions, or produce context-inconsistent code. Finally, ACI vulnerabilities in the Agent-Computer Interface through which agents interact with the environment represent an attack surface requiring permission controls and audit trails.

The literature proposes three categories of mitigation: model-centric (local deployment, fine-tuning), pipeline-centric (sandboxing, PII scrubbing, context minimisation), and algorithmic (mutation testing, output validation). TestForge incorporates elements from all three categories.

7.2 Privacy-by-Design in Test Generation

7.2.1 Local Execution as Primary Mitigation

The most effective privacy protection in TestForge is architectural: the entire platform (LLM inference, agent orchestration, and test execution) runs locally. By using Ollama (Ollama Inc. 2023) for LLM inference and Letta (formerly MemGPT) (Packer et al. 2023) (Docker) for agent memory, no application code, test data, or generated content leaves the user's machine.

This design directly addresses the data leakage concern identified in the literature. Unlike cloud-based approaches where code is transmitted to external servers, the local execution

model ensures that proprietary source code remains on the local machine, that golden test examples, which may contain domain-specific business logic, are not shared externally, that HTTP traffic captures containing authentication tokens or personal data are processed locally, and that generated tests referencing application internals are stored locally.

7.2.2 Data Minimisation

The platform follows privacy-by-design principles (Cavoukian 2011) and the principle of data minimisation at multiple levels. At the AST level, the Analyser Agent extracts only the structural patterns needed for the style guide (imports, fixture names, assertion types, and naming conventions) rather than sending entire test files to the LLM. The Observer Agent applies header filtering, stripping irrelevant HTTP headers (e.g., `Connection`, `Accept-Encoding`) from captured traffic before further processing. The Generator Agent practises context windowing by including only the most relevant golden examples in the LLM prompt rather than the entire test corpus.

7.2.3 Pseudonymisation and Identity Management

The auto-generated `conftest.py` uses dynamic email addresses (`f"test_{id(object())}@example.com"`) rather than realistic personal data, reducing the risk of PII appearing in test execution.

7.3 Security and System Integrity

7.3.1 Execution Isolation

Generated tests are executed in a subprocess with controlled environment variables. The Executor Agent writes tests to a temporary directory with limited scope, sets explicit environment variables (`BASE_URL`, `TESTFORGE_RUN`), and enforces execution timeouts (default: 60 seconds). Standard output and standard error are captured without granting the test process access to the parent process's state.

Future work includes Docker-based sandboxing for stronger isolation (see Chapter 8).

7.3.2 Prompt Injection Resilience

The platform's susceptibility to prompt injection attacks (Greshake et al. 2023) is limited by design. Golden examples are parsed via AST (Tree-sitter) rather than interpreted as LLM instructions, meaning that malicious comments in golden files are extracted as docstrings and not as prompt directives. Similarly, HTTP exchanges captured by the Observer are treated as structured data rather than as text to be interpreted by the LLM. The Generator Agent's system prompt includes explicit instructions about its role, constraining the LLM's behaviour even if user-supplied content contains injection attempts.

7.4 Ethical Considerations

7.4.1 Transparency

The platform provides full transparency into the generation process. The raw LLM response is preserved and available for inspection, and the Validator Agent provides per-test quality scores with explicit dimension breakdowns. Execution results include complete pytest output

(both stdout and stderr), and the Letta agent's conversation history is accessible, showing the reasoning behind generation decisions.

7.4.2 Human Oversight

TestForge is designed as a tool to assist developers, not to replace human judgement. Generated tests are presented for review before being added to a project's test suite, and the Validator Agent flags potential issues, such as missing assertions or suspicious patterns, for human attention. The conversational Letta interface further allows developers to guide, refine, and iterate on generated tests.

7.4.3 AI Tool Disclosure

This dissertation was produced with the assistance of AI tools (Claude, Ollama/llama3.1). AI was used for code generation, text drafting, and research exploration. All AI-generated content was reviewed, validated, and edited by the author. The Statement of Integrity at the front of this document provides full disclosure.

7.5 Regulatory Compliance

7.5.1 General Data Protection Regulation (GDPR)

The GDPR (European Parliament and Council 2016) establishes comprehensive requirements for the processing of personal data within the European Union. TestForge's architecture aligns with key GDPR principles:

Data minimisation (Art. 5(1)(c)) The platform processes only the data necessary for test generation, structural patterns from golden examples and endpoint schemas from traffic captures, rather than entire codebases.

Purpose limitation (Art. 5(1)(b)) Captured data is used exclusively for test generation and is not repurposed for model training, analytics, or other secondary uses.

Storage limitation (Art. 5(1)(e)) The `AppContext` stores only aggregated metadata (endpoint names, coverage percentages, run timestamps), not raw code or traffic data.

Data protection by design (Art. 25) Local execution eliminates cross-border data transfers, addressing one of GDPR's most operationally challenging requirements.

7.5.2 EU AI Act

The EU AI Act (European Parliament and Council 2024) (2024) classifies AI systems by risk level. An automated test generation tool is unlikely to fall into the "high-risk" category, as it does not directly affect fundamental rights, safety, or critical infrastructure. However, the platform's design nonetheless aligns with the Act's transparency requirements by providing full auditability of generated content and human oversight of the generation process.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

This section summarises the work accomplished in this iteration, reflecting on the prototype development, the experimental findings, and the overall progress toward the dissertation objective.

This dissertation set out to design, implement, and evaluate a multi-agent platform for automated test generation. The current iteration delivered a functional proof of concept, TestForge, that demonstrates the viability of combining LLM-powered test generation with AST-based pattern extraction from existing test files. The platform was built from the ground up as a modular Python package comprising six specialised agents organised into two complementary pipelines, with a persistent memory layer and full support for local LLM inference.

The Golden Examples pipeline, which was the primary focus of this iteration, was evaluated end-to-end against a Flask CRUD API. The Analyser agent successfully parsed 23 golden test functions using Tree-sitter, extracting a comprehensive style guide that captured the project's testing framework, HTTP client, fixture patterns, and assertion conventions. The Generator agent used this style guide together with the golden file source code as few-shot context to produce 9 new test functions that covered diverse testing categories including state integrity, boundary probing, business logic, type safety, and idempotency. Of these 9 tests, 4 passed and verified correct application behaviour, while 4 of the 5 failures exposed genuine application deficiencies (a validation ordering bug, missing input type validation, inconsistent error handling, and an incomplete search feature) that had not been detected by the original 23 hand-written tests. This 44% bug-finding rate demonstrates that the approach is capable of producing tests with real diagnostic value, going well beyond the trivial status code assertions that characterise most existing automated test generators.

The experiment also confirmed that local LLM execution using an 8B-parameter open-weight model (llama3.1:8b via Ollama) produces test code of sufficient quality for practical use, with all 9 generated tests being syntactically valid, properly structured, and executable by pytest. The generation latency of approximately 45 seconds for 9 tests, while slower than cloud-based inference, is acceptable for batch generation scenarios. Critically, the entire workflow ran on local infrastructure without transmitting any application code or test data to external services, validating the platform's suitability for organisations with strict data privacy requirements.

The multi-agent pipeline architecture proved effective for decomposing the test generation problem into manageable stages. The typed interfaces between agents, implemented as

Pydantic data models, prevented the cascading ambiguity issues that affect unstructured multi-agent communication and enabled independent development and debugging of each agent. The modular design also facilitated the implementation of the Black-Box Observer pipeline alongside the Golden Examples pipeline without requiring changes to the shared Generator, Executor, and Validator agents.

8.1.1 Summary of Contributions

The work completed in this iteration makes five concrete contributions. First, a systematic literature review of 55 studies following PRISMA methodology surveyed the state of the art in MAS-based software testing, LLM-driven test generation, and related security considerations, identifying research gaps that informed the platform design. Second, a golden examples approach using Tree-sitter AST parsing and few-shot prompting was implemented and evaluated, demonstrating the ability to generate tests that find real bugs by learning from existing high-quality test files. Third, the TestForge platform was developed as a modular, open-source prototype with six specialised agents, persistent memory via Letta, and fully local execution using open-weight models. Fourth, an experimental evaluation against a real-world application demonstrated a 44% pass rate and a 44% bug-finding rate, with a composite quality score of 0.65 out of 1.00. Fifth, a practical assessment of local LLM inference confirmed that effective test generation is feasible without cloud dependencies, addressing both privacy and cost concerns.

8.2 Limitations

This section acknowledges the limitations of the current iteration, distinguishing between constraints that stem from the scope of this delivery and those that represent fundamental challenges.

The most significant limitation is the scope of the experimental evaluation. The results are derived from a single target application (a Flask CRUD API with 7 endpoints) and generalisation to other application types, frameworks, authentication patterns, data models, and complexity levels has not been tested. Only the llama3.1:8b model (8 billion parameters) was evaluated; larger models such as llama3.1:70b, or cloud-hosted models like GPT-4 and Claude, may produce significantly better results with richer assertions and fewer incorrect assumptions. The results represent a single pipeline run without statistical analysis; multiple runs with different random seeds and statistical tests would provide more robust conclusions. No direct baseline comparison with existing test generation tools such as Pynguin or EvoSuite, nor with human-written tests for the same application, was performed in this iteration.

On the implementation side, the current prototype supports only Python with pytest and the requests library, and generalisation to other languages and testing frameworks requires additional development. Test execution relies on subprocess isolation rather than container-based sandboxing, which limits security guarantees when running untrusted generated code. The Black-Box Observer pipeline, while implemented at the architectural level, was not evaluated end-to-end in this iteration. Similarly, the persistent memory integration via Letta was validated at the infrastructure level but not tested across multiple sessions to measure its impact on progressive test quality improvement. These limitations define the scope of the next iteration.

8.3 Next Iteration and Future Work

This section outlines the work planned for the next iteration, which will complete the components that were not fully evaluated in the current delivery, as well as longer-term research directions that extend beyond the immediate scope of the dissertation.

8.3.1 Planned for the Next Iteration

Several components that were implemented but not fully evaluated in the current iteration are priorities for the next development cycle. The most immediate task is the end-to-end evaluation of the Black-Box Observer pipeline, which will involve capturing HTTP traffic from the target application via proxy interception, mapping the discovered endpoints, generating tests from the observed behaviour, and comparing the results against those obtained from the Golden Examples pipeline. This evaluation will determine whether traffic-based test generation produces tests of comparable quality to those generated from golden examples, and whether it can serve as a viable alternative when no reference tests are available.

The combined pipeline mode, which uses golden examples for style extraction and observer data for endpoint discovery, will also be evaluated against each individual pipeline to measure the benefit of complementary data sources. This comparison is central to the dissertation's dual-pipeline thesis and will provide empirical evidence on whether combining both approaches produces higher-quality test suites than either approach alone.

A multi-session longitudinal evaluation of the persistent memory system is also planned. This study will run the platform against the same target application across 5 to 10 sessions, measuring whether the ContextStore and Letta memory lead to progressively better tests as the platform accumulates knowledge about discovered endpoints, coverage gaps, and previously identified bugs. This evaluation will provide the empirical evidence needed to assess the value of persistent agent memory in automated test generation.

Additionally, a multi-model evaluation comparing generation quality across different LLMs (including llama3.1:70b, Mistral, and cloud-hosted GPT-4) will be conducted with controlled experiments to clarify the impact of model capability on test quality. A baseline comparison with existing test generation tools such as Pynguin and EvoSuite, as well as with human-written tests for the same application, will provide a more rigorous assessment of the platform's relative effectiveness.

8.3.2 Longer-Term Research Directions

Beyond the next iteration, several directions for future research emerge from this work. The Observer Agent could be extended to capture browser interactions via Playwright, enabling a UI-focused Generator to produce test scripts from recorded user flows rather than API calls alone. Support for intercepting LLM API calls within target applications would allow generating evaluation tests that assess response quality, consistency, and safety in AI-powered applications. On the security front, container-based test execution with resource limits, network isolation, and read-only filesystem mounts would provide production-grade sandboxing for running untrusted generated code.

A flakiness detection module that runs test suites multiple times would identify tests with inconsistent results, which is particularly important for generated tests that may depend on timing, ordering, or external state. The platform should also be tested against diverse

real-world applications including Django REST APIs, FastAPI services, microservice architectures, and applications with complex authentication schemes such as OAuth 2.0 and JWT tokens. Integration with CI/CD systems through plugins for GitHub Actions and GitLab CI would enable automatic test generation for new or modified endpoints on each pull request, embedding TestForge into the development workflow. Finally, formal benchmarking against established datasets such as HumanEval and SWE-bench adapted for testing, and comparison with state-of-the-art test generation tools under controlled conditions, would provide rigorous effectiveness evaluation and position the platform within the broader research landscape.

Bibliography

Ammann, Paul and Jeff Offutt (2016). "Introduction to Software Testing". In.

Atlidakis, Vaggelis, Patrice Godefroid, and Marina Polishchuk (2019). "RESTler: Stateful REST API Fuzzing". In: *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering (ICSE)*, pp. 748–758. doi: 10.1109/ICSE.2019.00083.

Austin, Jacob et al. (2021). "Program Synthesis with Large Language Models". In: *arXiv preprint arXiv:2108.07732*.

Bareiss, Patrick et al. (2022). "Code Generation Tools (Almost) for Free? A Study of Few-Shot, Pre-Trained Language Models on Code". In: *arXiv preprint arXiv:2206.01335*.

BerriAI (2023). *LiteLLM: Call all LLM APIs using the OpenAI format*. url: https://github.com/BerriAI/litellm.

Brown, Tom et al. (2020). "Language Models are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Vol. 33, pp. 1877–1901.

Brunsfeld, Max et al. (2018). *Tree-sitter: An incremental parsing system for programming tools*. url: https://tree-sitter.github.io/tree-sitter/.

Cavoukian, Ann (2011). "Privacy by Design: The 7 Foundational Principles". In: *Information and Privacy Commissioner of Ontario, Canada*. url: https://www.ipc.on.ca/wp-content/uploads/resources/7foundationalprinciples.pdf.

Chen, Mark et al. (2021). "Evaluating Large Language Models Trained on Code". In: *arXiv preprint arXiv:2107.03374*.

Colvin, Samuel (2017). *Pydantic: Data validation using Python type annotations*. url: https://docs.pydantic.dev/.

Cortesi, Aldo et al. (2010). *mitmproxy: A free and open source interactive HTTPS proxy*. url: https://mitmproxy.org/.

Dakhel, Arghavan Moradi et al. (2024). "Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing". In: *Information and Software Technology* 171, p. 107468. doi: 10.1016/j.infsof.2024.107468.

Decrop, Alix et al. (2024). "You Can REST Now: Automated Specification Inference and Black-Box Testing of RESTful APIs with Large Language Models". In: *arXiv preprint arXiv:2402.05102*.

European Parliament and Council (2016). *Regulation (EU) 2016/679 of the European Parliament and of the Council (General Data Protection Regulation)*. Official Journal of the European Union. url: https://eur-lex.europa.eu/eli/reg/2016/679/oj.

– (2024). *Regulation (EU) 2024/1689 laying down harmonised rules on artificial intelligence (Artificial Intelligence Act)*. Official Journal of the European Union. url: https://eur-lex.europa.eu/eli/reg/2024/1689/oj.

Fan, Angela et al. (2023). "Large Language Models for Software Engineering: Survey and Open Problems". In: *arXiv preprint arXiv:2310.03533*.

Feng, Zhangyin et al. (2020). "CodeBERT: A Pre-Trained Model for Programming and Natural Languages". In: *Findings of the Association for Computational Linguistics: EMNLP 2020*, pp. 1536–1547. doi: 10.18653/v1/2020.findings-emnlp.139.

Foster, Christopher et al. (2025). "Mutation-Guided LLM-based Test Generation at Meta". In: *arXiv preprint arXiv:2501.12862*.

Fraser, Gordon and Andrea Arcuri (2011). "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software". In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering (ESEC/FSE)*, pp. 416–419. doi: 10.1145/2025113.2025179.

Golmohammadi, Amid, Man Zhang, and Andrea Arcuri (2023). "Testing RESTful APIs: A Survey". In: *ACM Transactions on Software Engineering and Methodology* 33.1, pp. 1–41. doi: 10.1145/3617175.

Greshake, Kai et al. (2023). "Not what you've signed up for: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection". In: *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, pp. 79–90. doi: 10.1145/3605764.3623985.

Hong, Sirui et al. (2023). "MetaGPT: Meta Programming for A Multi-Agent Collaborative Framework". In: *arXiv preprint arXiv:2308.00352*.

Jia, Yue and Mark Harman (2011). "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5, pp. 649–678. doi: 10.1109/TSE.2010.62.

Jimenez, Carlos E. et al. (2024). "SWE-bench: Can Language Models Resolve Real-World GitHub Issues?" In: *Proceedings of the 12th International Conference on Learning Representations (ICLR)*.

Just, René, Darioush Jalali, and Michael D. Ernst (2014). "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA)*, pp. 437–440. doi: 10.1145/2610384.2628055.

Kang, Sungmin, Juyeon Yoon, and Shin Yoo (2023). "Large Language Models are Few-shot Testers: Exploring LLM-based General Bug Reproduction". In: *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pp. 2312–2323. doi: 10.1109/ICSE48619.2023.00194.

Kim, Myeongsoo, Saurabh Sinha, and Alessandro Orso (2025). "LlamaRestTest: Effective REST API Testing with Small Language Models". In: *Proceedings of the ACM on Software Engineering (FSE)*. doi: 10.1145/3715737.

Kim, Myeongsoo, Tyler Stennett, et al. (2024). "Leveraging Large Language Models to Improve REST API Testing". In: *Proceedings of the 46th International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. doi: 10.1145/3639476.3639769.

Kitchenham, Barbara and Stuart Charters (2007). *Guidelines for performing Systematic Literature Reviews in Software Engineering*. Technical Report EBSE-2007-01. Keele University and Durham University.

Krasner, Herb (2022). *The Cost of Poor Software Quality in the US: A 2022 Report*. Technical Report. Consortium for Information and Software Quality (CISQ).

Krekel, Holger et al. (2004). *pytest: helps you write better programs*. url: https://docs.pytest.org/.

Lemieux, Caroline et al. (2023). "CODAMOSA: Escaping Coverage Plateaus in Test Generation with Pre-trained Large Language Models". In: *Proceedings of the 45th International Conference on Software Engineering (ICSE)*, pp. 919–931. doi: 10.1109/ICSE48619.2023.00085.

Li, Raymond et al. (2023). "StarCoder: May the source be with you!" In: *arXiv preprint arXiv:2305.06161*.

Liu, Yi et al. (2022). "Morest: Model-based RESTful API Testing with Execution Feedback". In: *Proceedings of the 44th International Conference on Software Engineering (ICSE)*, pp. 1406–1417. doi: 10.1145/3510003.3510133.

Lukasczyk, Stephan, Florian Kroiß, and Gordon Fraser (2023). "An Empirical Study of Automated Unit Test Generation for Python". In: *Empirical Software Engineering* 28.2, p. 36. doi: 10.1007/s10664-022-10248-w.

Nashid, Noor, Mifta Sintaha, and Ali Mesbah (2023). "Retrieval-Based Prompt Selection for Code-Related Few-Shot Learning". In: *Proceedings of the 45th International Conference on Software Engineering (ICSE)*. doi: 10.1109/ICSE48619.2023.00205.

Ollama Inc. (2023). *Ollama: Get up and running with large language models locally*. url: https://ollama.com/.

Ouédraogo, Wendkuuni C., Abdoul Kader Kaboré, Yinghua Li, et al. (2024). "Large-scale, Independent and Comprehensive Study of the Power of LLMs for Test Case Generation". In: *arXiv preprint arXiv:2407.00225*.

Ouédraogo, Wendkuuni C., Abdoul Kader Kaboré, Haoye Tian, et al. (2024). "LLMs and Prompting for Unit Test Generation: A Large-Scale Evaluation". In: *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi: 10.1145/3691620.3695330.

Pacheco, Carlos et al. (2007). "Feedback-Directed Random Test Generation". In: *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pp. 75–84. doi: 10.1109/ICSE.2007.37.

Packer, Charles et al. (2023). "MemGPT: Towards LLMs as Operating Systems". In: *arXiv preprint arXiv:2310.08560*.

Page, Matthew J. et al. (2021). "The PRISMA 2020 statement: an updated guideline for reporting systematic reviews". In: *BMJ* 372, n71. doi: 10.1136/bmj.n71.

Park, Joon Sung et al. (2023). "Generative Agents: Interactive Simulacra of Human Behavior". In: *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology (UIST)*. doi: 10.1145/3586183.3606763.

Pizzorno, Juan Altmayer and Emery D. Berger (2024). "CoverUp: Coverage-Guided LLM-Based Test Generation". In: *arXiv preprint arXiv:2403.16218*.

Qian, Chen et al. (2024). "ChatDev: Communicative Agents for Software Development". In: *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (ACL)*, pp. 15174–15186.

Ramírez, Sebastián (2018). *FastAPI: Modern, fast web framework for building APIs with Python*. url: https://fastapi.tiangolo.com/.

Ronacher, Armin (2010). *Flask: A lightweight WSGI web application framework*. url: https://flask.palletsprojects.com/.

Rozière, Baptiste et al. (2023). "Code Llama: Open Foundation Models for Code". In: *arXiv preprint arXiv:2308.12950*.

Schäfer, Max et al. (2024). "An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation". In: *IEEE Transactions on Software Engineering* 50.1, pp. 85–105. doi: 10.1109/TSE.2023.3334955.

Shinn, Noah et al. (2023). "Reflexion: Language Agents with Verbal Reinforcement Learning". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 36.

Streamlit Inc. (2019). *Streamlit: A faster way to build and share data apps*. url: https://streamlit.io/.

Sumers, Theodore R. et al. (2023). "Cognitive Architectures for Language Agents". In: *arXiv preprint arXiv:2309.02427*.

Touvron, Hugo et al. (2023). "LLaMA: Open and Efficient Foundation Language Models". In: *arXiv preprint arXiv:2302.13971*.

W3C Web Performance Working Group (2012). *HTTP Archive (HAR) 1.2 Specification*. url: `https://w3c.github.io/web-performance/specs/HAR/Overview.html`.

Wei, Jason et al. (2022). "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *Advances in Neural Information Processing Systems*. Vol. 35, pp. 24824–24837.

Wooldridge, Michael (2009). *An Introduction to MultiAgent Systems*. 2nd. John Wiley & Sons. isbn: 978-0470519462.

Wu, Qingyun et al. (2023). "AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation". In: *arXiv preprint arXiv:2308.08155*.

Yang, John et al. (2024). "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: *Advances in Neural Information Processing Systems (NeurIPS)*. Vol. 37.

Yao, Shunyu et al. (2023). "ReAct: Synergizing Reasoning and Acting in Language Models". In: *Proceedings of the 11th International Conference on Learning Representations (ICLR)*.

Yuan, Zhiqiang et al. (2024). "No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation". In: *Proceedings of the ACM on Software Engineering (FSE)*. Vol. 1. doi: `10.1145/3660783`.

Zhang, Zeyu et al. (2025). "A Survey on the Memory Mechanism of Large Language Model based Agents". In: *ACM Transactions on Information Systems*. doi: `10.1145/3748302`.

Appendix A

Extended Code Listings

This appendix provides extended code listings for key components of the TestForge platform described in Chapter 5. Listings A.1–A.3 reflect the implemented prototype, while Listings A.4–A.7 present design-level specifications for security and deployment components planned for future iterations.

A.1 Configuration Schema

```python
from pydantic import BaseModel, Field
from typing import Dict, List, Optional
from enum import Enum

class LLMProvider(str, Enum):
    OPENAI = "openai"
    ANTHROPIC = "anthropic"
    OLLAMA = "ollama"

class LLMConfig(BaseModel):
    """Configuration for LLM provider."""
    provider: LLMProvider
    model: str
    api_key: Optional[str] = None
    base_url: Optional[str] = None
    temperature: float = Field(default=0.2, ge=0, le=2)
    max_tokens: int = Field(default=4096, ge=1)
    timeout: int = Field(default=60, ge=1)

class SandboxConfig(BaseModel):
    """Configuration for sandbox environments."""
    base_image: str = "python:3.11-slim"
    cpu_limit: int = Field(default=2, ge=1)
    memory_limit: str = "2g"
    timeout_seconds: int = Field(default=300, ge=1)
    network_mode: str = "none"
    allowed_hosts: List[str] = []

class SecurityConfig(BaseModel):
    """Security-related configuration."""
    enable_pii_scrubbing: bool = True
    pii_categories: List[str] = [
        "EMAIL_ADDRESS", "PHONE_NUMBER", "PERSON",
        "CREDIT_CARD", "IP_ADDRESS", "LOCATION"
    ]
    secret_patterns_file: Optional[str] = None
    audit_log_path: str = "logs/audit.jsonl"
```

```
    require_human_approval: bool = False
    approval_threshold: float = 0.8

class AgentConfig(BaseModel):
    """Configuration for individual agents."""
    agent_id: str
    role: str
    llm_config: LLMConfig
    tools: List[str] = []
    permissions: Dict[str, List[str]] = {}

class SystemConfig(BaseModel):
    """Top-level system configuration."""
    project_name: str
    agents: Dict[str, AgentConfig]
    sandbox: SandboxConfig
    security: SecurityConfig
    default_llm: LLMConfig
    output_directory: str = "tests/generated"
    log_level: str = "INFO"
```

Listing A.1: System configuration schema (config/schema.py)

A.2 LLM Gateway Implementation

```
from abc import ABC, abstractmethod
from typing import AsyncIterator, List, Optional
import asyncio
import hashlib
from functools import lru_cache

class LLMResponse(BaseModel):
    content: str
    model: str
    usage: Dict[str, int]
    finish_reason: str

class BaseLLMProvider(ABC):
    """Abstract base class for LLM providers."""

    @abstractmethod
    async def complete(
        self,
        messages: List[Dict],
        **kwargs
    ) -> LLMResponse:
        pass

    @abstractmethod
    async def stream(
        self,
        messages: List[Dict],
        **kwargs
    ) -> AsyncIterator[str]:
        pass

class OpenAIProvider(BaseLLMProvider):
    """OpenAI API provider implementation."""
```

```
34
35      def __init__(self, config: LLMConfig):
36          from openai import AsyncOpenAI
37          self.client = AsyncOpenAI(api_key=config.api_key)
38          self.config = config
39
40      async def complete(
41          self,
42          messages: List[Dict],
43          **kwargs
44      ) -> LLMResponse:
45          response = await self.client.chat.completions.create(
46              model=self.config.model,
47              messages=messages,
48              temperature=kwargs.get("temperature", self.config.
    temperature),
49              max_tokens=kwargs.get("max_tokens", self.config.max_tokens),
50          )
51          return LLMResponse(
52              content=response.choices[0].message.content,
53              model=response.model,
54              usage={
55                  "prompt_tokens": response.usage.prompt_tokens,
56                  "completion_tokens": response.usage.completion_tokens
57              },
58              finish_reason=response.choices[0].finish_reason
59          )
60
61  class LLMGateway:
62      """Central gateway for LLM interactions with caching and rate
    limiting."""
63
64      def __init__(
65          self,
66          config: SystemConfig,
67          security: "SecurityContext"
68      ):
69          self.config = config
70          self.security = security
71          self.providers: Dict[str, BaseLLMProvider] = {}
72          self.cache: Dict[str, LLMResponse] = {}
73          self.rate_limiter = asyncio.Semaphore(10)
74          self._init_providers()
75
76      def _init_providers(self):
77          """Initialize configured LLM providers."""
78          for agent_id, agent_config in self.config.agents.items():
79              llm_config = agent_config.llm_config
80              if llm_config.provider == LLMProvider.OPENAI:
81                  self.providers[agent_id] = OpenAIProvider(llm_config)
82              elif llm_config.provider == LLMProvider.ANTHROPIC:
83                  self.providers[agent_id] = AnthropicProvider(llm_config)
84              elif llm_config.provider == LLMProvider.OLLAMA:
85                  self.providers[agent_id] = OllamaProvider(llm_config)
86
87      async def complete(
88          self,
89          agent_id: str,
90          prompt: str,
```

```
        system: Optional[str] = None,
        use_cache: bool = True,
        **kwargs
    ) -> LLMResponse:
        """Execute LLM completion with security controls."""

        # Scrub PII from prompt
        scrubbed_prompt, scrub_log = self.security.scrub_pii(prompt)

        # Check cache
        cache_key = self._cache_key(agent_id, scrubbed_prompt, system)
        if use_cache and cache_key in self.cache:
            return self.cache[cache_key]

        # Rate limiting
        async with self.rate_limiter:
            messages = []
            if system:
                messages.append({"role": "system", "content": system})
            messages.append({"role": "user", "content": scrubbed_prompt
    })

            provider = self.providers.get(agent_id)
            if not provider:
                provider = self._get_default_provider()

            response = await provider.complete(messages, **kwargs)

        # Audit logging
        self.security.audit_log(
            event="llm_completion",
            agent_id=agent_id,
            prompt_hash=hashlib.sha256(scrubbed_prompt.encode()).
    hexdigest()[:16],
            response_length=len(response.content),
            tokens_used=sum(response.usage.values()),
            scrub_actions=len(scrub_log)
        )

        # Cache response
        if use_cache:
            self.cache[cache_key] = response

        return response

    def _cache_key(self, agent_id: str, prompt: str, system: str) -> str
    :
        content = f"{agent_id}:{system or ''}:{prompt}"
        return hashlib.sha256(content.encode()).hexdigest()
```

Listing A.2: LLM Gateway with provider abstraction (llm/gateway.py)

A.3 Prompt Templates

```
SYSTEM_PROMPT_TEST_GENERATION = """You are an expert software test
    engineer.
Your task is to generate high-quality pytest test cases.

```

```
Guidelines:
1. Each test should verify ONE specific behavior
2. Use descriptive names: test_<function>_<scenario>_<expected>
3. Follow the Arrange-Act-Assert pattern
4. Include edge cases and error conditions
5. Use appropriate fixtures and mocking
6. Generate meaningful assertions (not just `assert True`)

Output format:
Return a JSON array of test objects with this structure:
{
  "test_name": "test_function_scenario_expected",
  "test_code": "def test_function_scenario_expected():\n    ...",
  "target_function": "function_name",
  "rationale": "Why this test is important",
  "assertions": ["list", "of", "assertions"]
}
"""

TEST_GENERATION_PROMPT = """Generate pytest tests for the following code:

## Target Function
```python
{function_code}
```

Function Context
- Module: {module_name}
- Dependencies: {dependencies}
- Docstring: {docstring}

Additional Context
{additional_context}

Requirements
- Generate tests for normal behavior
- Generate tests for edge cases: {edge_cases}
- Generate tests for error handling
- Use these fixtures if needed: {available_fixtures}

Generate comprehensive tests as a JSON array.
"""

VALIDATION_PROMPT = """Review the following generated test for quality issues:

```python
{test_code}
```

Target function: {target_function}

Evaluate:
1. Does the test verify meaningful behavior?
2. Are assertions specific and non-trivial?
3. Is the test isolated and deterministic?
4. Does it follow testing best practices?
5. Are there any security concerns in the test code?
```

```
61
62 Return a JSON object with:
63 {
64   "quality_score": 0.0-1.0,
65   "issues": [{"severity": "...", "description": "..."}],
66   "recommendations": ["..."],
67   "approved": true/false
68 }
69 """
```

Listing A.3: Prompt templates for test generation (llm/prompts/test_generation.py)

A.4 Security Components (Design Specification)

The following listing presents the designed PII scrubber module, which combines regex-based secret detection with NER-based PII identification via Presidio. This component is planned for integration in a future iteration of the platform.

```
1  import re
2  from typing import Dict, List, Tuple
3  from presidio_analyzer import AnalyzerEngine, PatternRecognizer, Pattern
4  from presidio_anonymizer import AnonymizerEngine
5  from presidio_anonymizer.entities import OperatorConfig
6
7  class PIIScrubber:
8      """
9      Comprehensive PII scrubbing for code and text.
10     Combines regex patterns for code-specific secrets with
11     NER-based detection for general PII.
12     """
13
14     # Code-specific secret patterns
15     SECRET_PATTERNS = [
16         # API Keys
17         (r'(?i)(api[_-]?key|apikey)\s*[=:]\s*["\']?[\w-]{20,}["\']?',
18          '<API_KEY_REDACTED>'),
19         # Passwords
20         (r'(?i)(password|passwd|pwd|secret)\s*[=:]\s*["\']?[^\s
    "\']{8,}["\']?',
21          '<PASSWORD_REDACTED>'),
22         # AWS Keys
23         (r'AKIA[0-9A-Z]{16}', '<AWS_ACCESS_KEY_REDACTED>'),
24         (r'(?i)aws[_-]?secret[_-]?access[_-]?key\s*[=:]\s*["\']?[\w
    /+=]{40}["\']?',
25          '<AWS_SECRET_REDACTED>'),
26         # GitHub tokens
27         (r'ghp_[a-zA-Z0-9]{36}', '<GITHUB_TOKEN_REDACTED>'),
28         (r'gho_[a-zA-Z0-9]{36}', '<GITHUB_OAUTH_REDACTED>'),
29         # OpenAI keys
30         (r'sk-[a-zA-Z0-9]{48}', '<OPENAI_KEY_REDACTED>'),
31         # Generic tokens
32         (r'(?i)(bearer|token)\s+[a-zA-Z0-9\-_.]{20,}', '<
    BEARER_TOKEN_REDACTED>'),
33         # Private keys
34         (r'-----BEGIN (?:RSA |EC |DSA )?PRIVATE KEY-----[\s\S]*?-----END
     (?:RSA |EC |DSA )?PRIVATE KEY-----',
35          '<PRIVATE_KEY_REDACTED>'),
```

```
        # Connection strings
        (r'(?i)(mongodb|postgres|mysql|redis)://[^\s]+', '<CONNECTION_STRING_REDACTED>'),
        # JWT tokens
        (r'eyJ[a-zA-Z0-9_-]*\.eyJ[a-zA-Z0-9_-]*\.[a-zA-Z0-9_-]*', '<JWT_REDACTED>'),
    ]

    def __init__(self, config: Optional[SecurityConfig] = None):
        self.config = config or SecurityConfig()

        # Initialize Presidio
        self.analyzer = AnalyzerEngine()
        self.anonymizer = AnonymizerEngine()

        # Add custom recognizers for code patterns
        self._add_custom_recognizers()

    def _add_custom_recognizers(self):
        """Add custom pattern recognizers for code-specific secrets."""
        for i, (pattern, _) in enumerate(self.SECRET_PATTERNS):
            recognizer = PatternRecognizer(
                supported_entity=f"CODE_SECRET_{i}",
                patterns=[Pattern(name=f"secret_{i}", regex=pattern, score=0.9)]
            )
            self.analyzer.registry.add_recognizer(recognizer)

    def scrub(self, text: str) -> Tuple[str, List[Dict]]:
        """
        Scrub PII and secrets from text.

        Args:
            text: Input text potentially containing PII

        Returns:
            Tuple of (scrubbed_text, scrub_log)
        """
        scrub_log = []
        cleaned = text

        # First pass: regex-based secret detection (faster)
        for pattern, replacement in self.SECRET_PATTERNS:
            matches = list(re.finditer(pattern, cleaned))
            if matches:
                cleaned = re.sub(pattern, replacement, cleaned)
                scrub_log.append({
                    "type": "code_secret",
                    "pattern": pattern[:30] + "...",
                    "count": len(matches),
                    "replacement": replacement
                })

        # Second pass: Presidio NER-based detection
        if self.config.pii_categories:
            results = self.analyzer.analyze(
                cleaned,
                entities=self.config.pii_categories,
                language="en"
```

```
            )

            if results:
                # Configure anonymization operators
                operators = {
                    "EMAIL_ADDRESS": OperatorConfig("replace", {"new_value": "<EMAIL_REDACTED>"}),
                    "PHONE_NUMBER": OperatorConfig("replace", {"new_value": "<PHONE_REDACTED>"}),
                    "PERSON": OperatorConfig("replace", {"new_value": "<PERSON_REDACTED>"}),
                    "CREDIT_CARD": OperatorConfig("replace", {"new_value": "<CC_REDACTED>"}),
                    "IP_ADDRESS": OperatorConfig("replace", {"new_value": "<IP_REDACTED>"}),
                    "LOCATION": OperatorConfig("replace", {"new_value": "<LOCATION_REDACTED>"}),
                }

                anonymized = self.anonymizer.anonymize(
                    cleaned,
                    results,
                    operators=operators
                )
                cleaned = anonymized.text

                for result in results:
                    scrub_log.append({
                        "type": result.entity_type,
                        "score": result.score,
                        "start": result.start,
                        "end": result.end
                    })

        return cleaned, scrub_log

    def scan_only(self, text: str) -> List[Dict]:
        """Scan for PII without modifying text."""
        _, log = self.scrub(text)
        return log
```

Listing A.4: PII Scrubber design specification (security/scrubber.py)

A.5 Docker Sandbox Configuration (Design Specification)

The following listings present the designed Docker sandbox configuration for secure test execution. The current prototype uses subprocess-based isolation; container-based sandboxing is planned for a future iteration.

```
# Secure Python sandbox for test execution
FROM python:3.11-slim-bookworm

# Security: Create non-root user
RUN groupadd -r sandbox && useradd -r -g sandbox sandbox

# Install minimal dependencies
RUN apt-get update && apt-get install -y --no-install-recommends \
    git \
```

```
    && rm -rf /var/lib/apt/lists/* \
    && apt-get clean

# Security: Remove unnecessary tools
RUN rm -rf /usr/bin/wget /usr/bin/curl 2>/dev/null || true

# Create workspace with proper permissions
RUN mkdir -p /workspace /home/sandbox && \
    chown -R sandbox:sandbox /workspace /home/sandbox

# Install Python testing tools
COPY requirements-sandbox.txt /tmp/
RUN pip install --no-cache-dir -r /tmp/requirements-sandbox.txt && \
    rm /tmp/requirements-sandbox.txt

# Security: Switch to non-root user
USER sandbox
WORKDIR /workspace

# Security: Set restrictive umask
ENV UMASK=077

# Default command (overridden at runtime)
CMD ["python", "--version"]
```

Listing A.5: Dockerfile for secure sandbox environment (docker/Dockerfile.sandbox)

```
{
  "defaultAction": "SCMP_ACT_ERRNO",
  "architectures": ["SCMP_ARCH_X86_64"],
  "syscalls": [
    {
      "names": [
        "read", "write", "open", "close", "stat", "fstat",
        "lstat", "poll", "lseek", "mmap", "mprotect", "munmap",
        "brk", "rt_sigaction", "rt_sigprocmask", "rt_sigreturn",
        "ioctl", "access", "pipe", "select", "sched_yield",
        "mremap", "msync", "mincore", "madvise", "dup", "dup2",
        "nanosleep", "getpid", "getuid", "getgid", "geteuid",
        "getegid", "getppid", "getpgrp", "setsid", "getgroups",
        "uname", "fcntl", "flock", "fsync", "fdatasync",
        "truncate", "ftruncate", "getcwd", "chdir", "rename",
        "mkdir", "rmdir", "link", "unlink", "symlink", "readlink",
        "chmod", "fchmod", "chown", "fchown", "umask", "getrlimit",
        "getrusage", "times", "clock_gettime", "clock_getres",
        "exit", "exit_group", "wait4", "kill", "clone", "fork",
        "vfork", "execve", "arch_prctl", "set_tid_address",
        "set_robust_list", "futex", "sched_getaffinity",
        "openat", "newfstatat", "readlinkat", "faccessat",
        "pread64", "pwrite64", "getrandom"
      ],
      "action": "SCMP_ACT_ALLOW"
    }
  ]
}
```

Listing A.6: Seccomp security profile (docker/seccomp-profile.json)

A.6 CLI Implementation (Design Specification)

The following listing presents a designed command-line interface using Click and Rich. The current prototype exposes its CLI through a simpler interface; this specification targets a future release.

```
import click
import asyncio
from pathlib import Path
from rich.console import Console
from rich.progress import Progress, SpinnerColumn, TextColumn

console = Console()

@click.group()
@click.version_option(version="0.1.0")
def cli():
    """MAS Testing - Secure Multi-Agent Software Testing System"""
    pass

@cli.command()
@click.argument('project_path', type=click.Path(exists=True))
@click.option('--requirements', '-r', default="",
              help='Testing requirements in natural language')
@click.option('--output', '-o', default='tests/generated',
              help='Output directory for generated tests')
@click.option('--config', '-c', type=click.Path(exists=True),
              help='Path to configuration file')
@click.option('--model', default='gpt-4-turbo',
              help='LLM model to use')
@click.option('--local', is_flag=True,
              help='Use local LLM (Ollama) for privacy')
@click.option('--dry-run', is_flag=True,
              help='Analyze without generating tests')
@click.option('--verbose', '-v', is_flag=True,
              help='Enable verbose output')
def generate(project_path, requirements, output, config, model, local,
    dry_run, verbose):
    """Generate tests for a Python project."""

    console.print(f"[bold blue]MAS Testing[/bold blue] - Secure Test
    Generation")
    console.print(f"Project: {project_path}")

    # Load configuration
    system_config = load_config(config) if config else default_config()

    if local:
        system_config.default_llm.provider = LLMProvider.OLLAMA
        system_config.default_llm.model = "codellama:34b"
        console.print("[green]Using local LLM for privacy[/green]")
    else:
        system_config.default_llm.model = model

    # Initialize system
    with Progress(
        SpinnerColumn(),
        TextColumn("[progress.description]{task.description}"),
        console=console
```

```
    ) as progress:

        task = progress.add_task("Initializing agents...", total=None)
        workflow = TestingWorkflow.from_config(system_config)

        if dry_run:
            progress.update(task, description="Analyzing project...")
            analysis = asyncio.run(workflow.analyze_only(project_path))
            display_analysis(analysis)
            return

        progress.update(task, description="Generating tests...")
        result = asyncio.run(workflow.run(project_path, requirements))

    # Display results
    if result.status == "completed":
        output_path = Path(output)
        output_path.mkdir(parents=True, exist_ok=True)

        for test in result.tests:
            test_file = output_path / f"test_{test.target_function}.py"
            test_file.write_text(test.test_code)

        console.print(f"\n[green]Success![/green] Generated {len(result.tests)} tests")
        console.print(f"Output directory: {output_path}")

        # Summary table
        display_summary(result)
    else:
        console.print(f"\n[red]Failed:[/red] {result.reason}")
        if result.security_findings:
            console.print("\n[yellow]Security issues found:[/yellow]")
            for finding in result.security_findings:
                console.print(f"  - {finding.description}")

@cli.command()
@click.argument('project_path', type=click.Path(exists=True))
@click.option('--output', '-o', default='security-report.json',
              help='Output file for security report')
def audit(project_path, output):
    """Run security audit on a project."""
    console.print("[bold]Running security audit...[/bold]")

    # Initialize security agent only
    security_agent = SecurityAgent.from_default_config()

    with Progress(SpinnerColumn(), TextColumn("{task.description}")) as progress:
        task = progress.add_task("Scanning...", total=None)
        result = asyncio.run(security_agent.full_scan(project_path))

    # Write report
    Path(output).write_text(result.model_dump_json(indent=2))
    console.print(f"\n[green]Audit complete.[/green] Report: {output}")
    console.print(f"Risk level: {result.overall_risk}")

if __name__ == '__main__':
    cli()
```

Listing A.7: Command-line interface design specification (integration/cli.py)

Appendix B

Detailed Experimental Results

This appendix provides the complete experimental data from the Golden Examples pipeline evaluation described in Chapter 6, including the generated test code, per-test execution details, and the validator output.

B.1 Target Application Endpoints

Table B.1 lists all endpoints of the Flask CRUD API used as the target application.

Table B.1: Target application endpoints and their expected behaviour

Method	Path	Success	Notes
GET	/api/health	200	Returns `{"status": "healthy"}`
GET	/api/users	200	Returns list of all users
POST	/api/users	201	Requires `name` and `email`; returns 400 if missing, 409 if duplicate email
GET	/api/users/{id}	200	Returns 404 if not found
PUT	/api/users/{id}	200	Partial update; returns 404 if not found
DELETE	/api/users/{id}	200	Returns 404 if not found
GET	/api/users/search	200	Query parameter `q`; returns 400 if missing

B.2 Golden Test Examples Summary

Table B.2 summarises the three golden test files provided as input to the pipeline.

Table B.2: Golden test files: per-file breakdown

File	Tests	Assertions	Endpoints	Focus
`test_users_golden_1.py`	7	24	4	Health, create, get, list
`test_users_golden_2.py`	10	31	5	Update, delete, search
`test_users_golden_3.py`	6	19	3	Edge cases, duplicates
Total	**23**	**74**	**7**	

B.3 Analyser Agent Output

The style guide extracted by the Analyser Agent from the 23 golden tests is summarised in Table B.3.

Table B.3: Extracted style guide details

Attribute	Value
Framework	`pytest`
HTTP Client	`requests`
Common Imports	`pytest`, `requests`
Common Fixtures	`base_url`, `created_user`, `sample_user`
Naming Convention	`test_<action>_<scenario>`
Average Assertions/Test	3.2
Docstring Coverage	100% (23/23 tests)
HTTP Methods Used	GET (12), POST (6), PUT (3), DELETE (2)

B.4 Generated Test Details

Table B.4 provides per-test details for the 9 tests generated by the Golden Examples pipeline, including the number of assertions, endpoints targeted, and failure details.

Table B.4: Per-test details of the 9 generated tests

Test Name	Category	Asserts	Result	Failure Reason
`test_data_round_trip`	State Integrity	5	PASSED	—
`test_partial_update_safety`	State Integrity	4	PASSED	—
`test_delete_consistency`	State Integrity	4	PASSED	—
`test_duplicate_handling`	Business Logic	3	PASSED	—
`test_boundary_values`	Boundary Probing	4	FAILED	Expected 400 for empty strings; got 409 (validation ordering)
`test_type_confusion`	Type Safety	3	FAILED	Expected 400 for non-integer id; got 201 (missing type check)
`test_ordering_filtering`	Ordering/Filtering	3	FAILED	Expected filter functionality not implemented
`test_error_message_quality`	Error Quality	4	FAILED	Expected 400 for non-JSON; got 415 (inconsistent error code)
`test_idempotency`	Idempotency	3	FAILED	State pollution from prior test caused 409 conflict

B.5 Validator Agent Scores

Table B.5 presents the quality scores assigned by the Validator Agent to each generated test.

Table B.5: Validator Agent quality scores per test

Test Name	Assertion	Coverage	Readability	Execution	Overall
test_data_round_trip	0.90	0.75	0.85	1.00	0.88
test_partial_update_safety	0.80	0.50	0.85	1.00	0.79
test_delete_consistency	0.80	0.75	0.80	1.00	0.84
test_duplicate_handling	0.70	0.50	0.80	1.00	0.75
test_boundary_values	0.80	0.50	0.80	0.00	0.53
test_type_confusion	0.70	0.50	0.75	0.00	0.49
test_ordering_filtering	0.70	0.50	0.80	0.00	0.50
test_error_message_quality	0.80	0.50	0.85	0.00	0.54
test_idempotency	0.70	0.75	0.80	0.00	0.56
Average	**0.77**	**0.58**	**0.81**	**0.44**	**0.65**

B.6 Bug Classification

Table B.6 classifies each failing test by whether it reveals a genuine application deficiency or a test-side issue.

Table B.6: Classification of test failures

Test Name	Type	Severity	Description
test_boundary_values	App bug	Medium	Duplicate detection runs before field validation; empty strings accepted as valid names
test_type_confusion	App bug	High	No type validation on input fields; application accepts arbitrary types for id
test_ordering_filtering	Missing feature	Low	Search/filter functionality not fully implemented
test_error_message_quality	App bug	Medium	Inconsistent error codes: returns 415 instead of 400 for content-type errors
test_idempotency	Test isolation	Low	State leakage between tests; not an application deficiency

B.7 Endpoint Coverage Analysis

Table B.7 shows which endpoints were targeted by the generated tests.

B.8 Insightful Category Coverage

Table B.8 shows the distribution of generated tests across the eight insightful test categories defined in the Generator Agent's prompt.

Note: The three uncovered categories (concurrency, authorisation, data leakage) are more relevant to applications with authentication and concurrent access patterns. The target Flask CRUD API uses in-memory storage without authentication, which explains the generator's focus on other categories.

Table B.7: Endpoint coverage: golden tests vs. generated tests

Method	Path	Golden (23 tests)	Generated (9 tests)
GET	/api/health	✓	—
GET	/api/users	✓	—
POST	/api/users	✓	✓
GET	/api/users/{id}	✓	✓
PUT	/api/users/{id}	✓	✓
DELETE	/api/users/{id}	✓	✓
GET	/api/users/search	✓	✓
Coverage		**7/7 (100%)**	**5/7 (71%)**

Table B.8: Test category coverage

Category	Tests	Coverage
State Integrity	3	✓
Boundary Probing	1	✓
Business Logic	1	✓
Error Quality	1	✓
Idempotency	1	✓
Concurrency Hints	0	—
Authorisation Boundaries	0	—
Data Leakage	0	—
Total	**9**	**4/8 predefined**