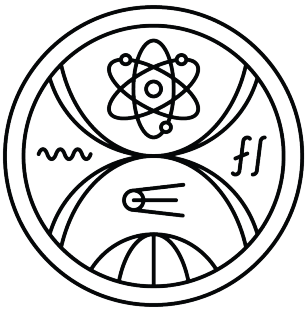
COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS

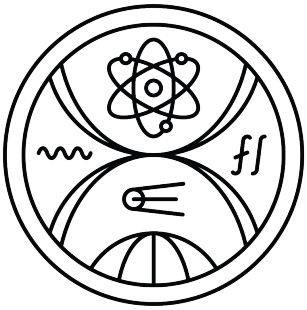


GENEROVANIE TESTOV SOFTVÉROVÝCH SYSTÉMOV

Diplomová Práca

2025 Bc. Ákos Czére

COMENIUS UNIVERSITY IN BRATISLAVA
FACULTY OF MATHEMATICS PHYSICS AND INFORMATICS



SOFTWARE TEST GENERATION

Diploma Thesis

Study program:	Applied informatics
Branch of study:	Applied informatics
Department:	Department of Applied Informatics
Supervisor:	doc. Ing. Ivan Polášek, PhD.

Bratislava, 2025 Bc. Ákos Czére

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bc. Ákos Czére
Study programme:	Applied Computer Science (Single degree study, master II. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Diploma Thesis
Language of Thesis:	Slovak
Secondary language:	English

Title: Generovanie testov softvérových systémov
Test generation for software systems

Annotation: Jednou z dôležitých etáp pri vývoji softvérových systémov je aj ich testovanie.

Aim: Navrhnite postup generovania testov (v podobe try – catch, assert(), alebo standardnych podmienok a podobne) k existujúcemu zdrojovému kódu pomocou AI s využitím sady podobných kolekcií už existujúcich testov alebo šablón.
Navrhnite kontrolu pokrytia testov podľa zdrojového kódu alebo prípadov použitia a vyhodnoťte ich účinnosť.

Supervisor:	doc. Ing. Ivan Polášek, PhD.
Department:	FMFI.KAI - Department of Applied Informatics
Head of department:	doc. RNDr. Tatiana Jajcayová, PhD.
Assigned:	06.11.2024
Approved:	08.11.2024

prof. RNDr. Roman Ďurikovič, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Ákos Czére
Študijný program: aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: diplomová
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Generovanie testov softvérových systémov
Test generation for software systems

Anotácia: Jednou z dôležitých etáp pri vývoji softvérových systémov je aj ich testovanie.

Cieľ: Navrhnite postup generovania testov (v podobe try – catch, assert(), alebo standardnych podmienok a podobne) k existujúcemu zdrojovému kódu pomocou AI s využitím sady podobných kolekcií už existujúcich testov alebo šablón.
Navrhnite kontrolu pokrytia testov podľa zdrojového kódu alebo prípadov použitia a vyhodnoťte ich účinnosť.

Vedúci: doc. Ing. Ivan Polášek, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: doc. RNDr. Tatiana Jajcayová, PhD.

Dátum zadania: 06.11.2024

Dátum schválenia: 08.11.2024 prof. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

. . . . . .

Bratislava, 2025 Bc. Ákos Czére

Acknowledgement

Abstract

The aim of this thesis is to design and implement a method for the automated generation of unit tests for existing source code using artificial intelligence. The proposed solution leverages a set of templates or collections of existing tests as a reference, with tests being generated in the form of try–catch blocks, using the assert() function, or standard conditional statements. The work also includes the design and implementation of a mechanism for test coverage analysis based on source code inspection or use case analysis, along with an evaluation of the effectiveness of the proposed solution in terms of test quality and completeness. The practical part of the thesis involves the development of a prototype utilizing techniques such as fine-tuning of large language models (LLMs) and Retrieval-Augmented Generation (RAG), with the aim of improving the accuracy and relevance of the generated tests. The results of the thesis demonstrate the potential of these methods in supporting automated testing within the field of software engineering.

Keywords: Large Language Model, unit test generation, fine-tuning, retrieval-augmented generation

Abstrakt

Cieľom tejto diplomovej práce je navrhnúť a realizovať postup automatizovaného generovania unit testov pre existujúci zdrojový kód pomocou umelej inteligencie. Navrhované riešenie využíva sadu šablón alebo kolekcií už existujúcich testov ako referenčný základ, pričom testy sú generované vo forme blokov try–catch, pomocou funkcie assert() alebo pomocou štandardných podmienok. Súčasťou práce je aj návrh a implementácia mechanizmu kontroly pokrytia testov na základe analýzy zdrojového kódu a prípadov použitia, ako aj vyhodnotenie účinnosti navrhnutého riešenia z hľadiska kvality a úplnosti testovania. Praktická časť práce zahŕňa vývoj prototypu využívajúceho techniky ako fine-tuning veľkých jazykových modelov (LLM) a Retrieval-Augmented Generation (RAG), s cieľom zlepšiť presnosť a relevanciu generovaných testov. Výsledky práce ukazujú potenciál týchto metód pri podpore automatizovaného testovania v softvérovom inžinierstve.

Kľúčové slová: Large Language Model, unit test generation, fine-tuning, retrieval-augmented generation

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

In today's world, where more and more parts of our lives are becoming digitalized and automated, it is no surprise that developers are also trying to find ways to make their jobs easier. One way to accomplish this goal is to automate the repetitive, boring, but also crucial task of software testing. This, however, is not an easy task even for the most sophisticated systems, as it requires knowledge not only of the system itself but also of every other system it depends on.

There are a number of approaches that modern systems already use to generate different kinds of high-quality software tests, such as evolutionary search approaches that evolve whole test suites, like EvoSuite [2]. Recently, however, large language models have also shown promising signs of being suitable for this purpose[12]. Their strengths include understanding source code, strong code-generation capabilities, and, more recently, the ability to reason about the task and the source code at hand.

Unfortunately, they also have major drawbacks. Although existing solutions based on large language models can generate tests that are syntactically valid, they often struggle to create tests that achieve high coverage, quality, usability, and logical correctness for real-world systems. They can also introduce hallucinated details about the system under test[12].

Our hypothesis for why these shortcomings occur is that, no matter how large a model is, it will struggle with the amount of context it needs to be aware of to generate relevant and meaningful tests. Our proposed solution is not to rely on a single, general-purpose model that is prompted with multiple queries, but instead to create a system of connected, specialized agents. Each agent has a clearly defined role, such as summarizing a piece of the source code, retrieving information about the relationships between components of the system under test, or reviewing the generated tests and rewriting them if necessary. Another way the system aims to improve agent

performance is by defining clear templates for the inputs and outputs of every agent.

By splitting the task of test generation into multiple smaller parts, we hope to retain as much context of the system under test as possible while using the least number of tokens needed. To further improve the agents' performance, they are equipped with Retrieval-Augmented Generation techniques, ranging from traditional RAG to the use of knowledge graphs based on Microsoft's GraphRAG framework.

1.2 Problem Statement

Despite the capabilities of modern large language models, generating coherent, meaningful tests still shows to be a problem for them. Current approaches often run into several common problems:

- **Missing context:** The model may not fully understand how different functions, classes, and modules relate to each other.
- **Incorrect details:** Tests sometimes contain wrong assumptions, nonexistent method calls, or made-up logic.
- **No feedback process:** If the test is wrong, the model usually does not get a chance to fix it.
- **Scaling issues:** For larger projects, simple prompting becomes impractical because the model cannot fit all the relevant information into a single prompt.

These limitations significantly impact the application of LLMs in SE, and also highlight the need for expert developeers to critically refine and validate LLMgenerated code for accuracy and security[6].

The main question this thesis aims to answer is:

> *How can we build a dependable, step-by-step LLM-based system that produces useful and correct tests for software codebases?*

To address these challenges, researchers have begun developing LLM-based agents. At their core, these agents are still large language models, which enables them to harness their core reasoning and language processing capabilities, but also utilize a wide range of external tools, API calls, and even knowledge sources with the help of retrieval augmented generation. These extra capabilities enable the agents to interact with their environment, access up-to-date information, and perform complex tasks leading to a more robust, adaptive, and autonomous operations.

To addresses the aforementioned problems, the proposed system utilizes a structured multi-agent workflow, supported by graph-based retrieval and iterative validation.

1.3 Research Goals and Contributions

The main goal of this thesis is to design and build a system that can automatically generate meaningful tests for Python projects. To achieve this, the following objectives were defined:

- Create a pipeline of LLM agents using LangGraph, where each node has a specific task.
- Analyze source code files and produce summaries that capture the most important information.
- Use Microsoft GraphRAG (running on Ollama) to find related classes, functions, and modules.
- Generate test-case scenarios in a structured, predefined format.
- Write tests based on those scenarios using an LLM.
- Validate the generated tests and automatically rewrite them if they are incorrect.
- Combine and save all final tests into one output.

The contributions of this work include:

- A complete eight-node LangGraph workflow for test generation.
- A file summarization approach that helps the system understand the structure of the project.
- An integration of GraphRAG that allows local, graph-based retrieval of related code.
- A structured test-scenario format that guides the test generation process.
- An iterative validation and rewriting loop that improves reliability.
- An extensible design that can be expanded with new retrieval techniques or agent types.

1.4 Scope and Limitations

This thesis focuses on generating unit tests for Java projects. The system works on local repositories and uses Ollama to run LLMs on the user's machine. It is built to be modular so that different models or retrieval methods can be added in the future.

However, the system also has several limitations:

- **Model accuracy:** The tests are only as good as the agents used in the pipeline.
- **Complex codebases:** Some dynamic, framework-heavy, or highly abstract code may still be difficult to test automatically.
- **Performance:** Building the GraphRAG index can take time, especially for larger repositories.
- **Evaluation scope:** The system is tested on a limited set of projects.
- **No full tooling integration:** It is not yet connected to continuous integration systems or IDEs.

These points help set realistic expectations and highlight areas for future work.

1.5 Thesis Structure

The rest of this thesis is organized as follows:

- **Chapter 2** gives an overview of related work, including software testing, LLM-based code generation, RAG techniques, and LangGraph.
- **Chapter 3** describes the architecture of the system and explains how each node in the pipeline works.
- **Chapter 4** discusses the implementation details, such as prompt design, model configuration, and agent coordination.
- **Chapter 5** presents possible improvements and future extensions of the system.
- **Chapter 6** evaluates the system on example codebases and analyzes the results.
- **Chapter 7** concludes the thesis by summarizing the findings and outlining the main limitations.

This structure guides the reader from the motivation and background to the design, implementation, evaluation, and final reflections on the developed test generation system.

Chapter 2

Background and Related Work

This chapter focuses on the key areas that this thesis consists of:

1. The landscape of automated software testing
2. Large language model approaches to code and test generation
3. Retrieval-augmented generation techniques, including graph-based RAG
4. Agent management frameworks, focusing on LangGraph, as this is the framework that was used in our system.

The chapter finishes with a comparison of three representative systems from the literature, namely AgoneTest[8], ASTER[10], and GenUTest[11], while highlighting how they relate to the design choices made for the propsed system.

2.1 Overview of Automated Software Testing

One of the most crucial stages in software development is software testing. This stage aims to ensure long-term reliability by detecting defects and logical errors in the system early in the development life cycle. Conventional approaches include manually written unit, integration, and system tests, while examples of more advanced techniques could be symbolic execution or search-based testing.

In recent years, the demand for automated test generation has increased significantly due to the increasing complexity of software systems and the endless need for further integration and delivery of new features.

To minimize the amount of human effort needed to write software tests, many automated test generation approaches have emerged, however the task of generating complete and logically correct tests with high code coverage remains a difficult challenge. This is especially true for systems with multiple, possibly proprietary dependencies or dynamic behavior.

2.2 LLM-based code generation and test generation

Large Language Models have demonstrated strong capabilities in code understanding, code generation, and reasoning about software. Modern models can interpret function signatures, generate syntactically correct code, and even make suggestions for improving existing programs, which make them natural candidates for cutting edge test generation tools. This is exactly the subject of many recent studies, that have explored the possibility of unit test generation with large language models, for example Andrzejewski et al.[1] evaluate the performance of various models, (namely Claude, Llama and Mistral), and prompts for generating Python unit tests of various difficulty, then compares it to human written tests. The paper highlights common issues such as producing effective tests only for very easy and easy code cases while their performance drops considerably when facing medium or hard problems. Based on the same paper's conclusion, this statement also holds true for not only execution success, but code coverage, too. Increasing prompt complexity did not seem to reliably improve test quality, either, which suggests that more context does not always lead to better results, which aligns with the broader perception that large laguage models, although promising, are still unreliable when used in a single step or single agent manner.

The main problems in LLM-based test generation identified in the literature include:

- Insufficient global context about the codebase.
- Limited ability to extract and integrate cross-file or project-level knowledge.
- Lack of iterative validation and repair.
- Hallucinations or incorrect assumptions about the system under test.

These limitations motivate the development of more structured, multi-step, or multi-agent approaches, such as the one proposed in this thesis.

2.3 Retrieval-Augmented Generation

Retrieval-Augmented Generation, or RAG for short, is a technique that aims to improve the reliability and accuracy of the outputs generated by large language models by supplying them with non-parameterized (external), relevant information retrieved from vector stores, databases, or knowledge graphs. RAG has proven effective for question answering, documentation assistance, and code analysis tasks, primarily because it reduces hallucinations and gives the model access to ground-truth context[3].

Recently, graph-based retrieval has been introduced as an extension of classical RAG. One such approach is Microsoft's GraphRAG framework, which constructs a

knowledge graph from documents and uses graph traversal to identify the most relevant entities and relationships. This is especially useful in software projects, where functions, classes, files, and modules have non-trivial dependencies. The system proposed in this thesis makes use of GraphRAG to maintain a structured understanding of the codebase and to support context-aware test generation.

2.4 Graph-based RAG and Microsoft GraphRAG

2.4.1 How GraphRAG Works

GraphRAG is an extention of the traditional retrieval-augmented generation techniques. The main difference is, that GraphRAG creates a so called knowledge graph over the domain data (in our case, examples of this data could be source code summaries), instead of using a collection of text chunks. In GraphRAG, entities like classes, methods and modules are represented as nodes of a graph and the relationships between them, for example function calls, imports, inheritance, dependencies and so on, become edges between these nodes. This way information is stored in a structured representation, that highlights the connections between a system's components, which is especially useful for complex codebases where classes and their methods can be referenced across multiple modules.

When a query like "generate test scenarios for classX.methodY" is sent to GraphRag, it performs graph based retrieval, meaning that rather than selecting some semantically similar chunks of text, it traverses the knowledge graph to collect a relevant subgraph. This subgraph, that according to the large language model used by GraphRag contains related classes, methods and other dependencies, is then converted into a prompt context for the large language model that then generates output based on this enriched, relational context. [5].

2.4.2 GraphRAG vs. Traditional RAG

Traditional RAG approaches typically treat the knowledge base as just a set of documents or code/text chunks. Retrieval is based on similarity or keyword matching, which works well when relevant information is concentrated in a single location. However, when context is distributed, for example in our case, where the software is comprised of interactions between multiple classes, modules, or dynamically instantiated objects, similarity based retrieval may miss important dependencies.

In contrast, GraphRAG's relational structure allows multi-hop reasoning and context aggregation, meaning that by traversing edges and collecting connected entities, GraphRAG can deduce relevant dependencies, like helper classes, utility functions or

Figure 2.1: Visualization of a graph created by GraphRag. The graph was created from source code summaries of the Asynchronous Database Connectivity in Java repository (https://github.com/mheath/adbcj/tree/master) generated from the system proposed in this thesis

inherited behavior, enabling a more holistic context. This is particularly helpful for software test generation, where tests may need to consider interactions across multiple code elements.

That said, GraphRAG can impose greater complexity compared to traditional RAG, as the graph constructed (entity and relation extraction, indexing), subgraph selection and summarization must be done carefully to avoid over-loading the LLM prompt, and retrieval latency or token usage may increase when context is large. Empirical results on GraphRAG systems show that while it often outperforms RAG on complex, multi-hop reasoning tasks, it sometimes lags on simpler tasks due to overhead or unnecessary graph traversal [5].

2.4.3 Strengths and Limitations of GraphRAG for Code/Test Generation

Strengths:

- **Relational context awareness:** GraphRAG captures relations like class inheritance, method calls, module dependencies — essential for understanding how parts of a codebase interact. This helps the LLM generate tests that reflect realistic code interactions, not just isolated methods.

- **Multi-hop retrieval and reasoning:** GraphRAG can surface indirect dependencies (e.g., utility modules, helper functions) that plain text retrieval might miss, improving coverage and reducing chance of missing relevant context.
- **Reduced hallucination risk:** Because the context is grounded in an explicit graph derived from actual code metadata, the LLM is less likely to invent non-existent functions or relationships.
- **Scalable to larger codebases:** Instead of embedding entire repositories or very large files into a prompt, GraphRAG allows selective, relation-driven retrieval, helping stay within token limits while preserving relevant context.

Limitations and Challenges:

- **Graph construction overhead:** Building a reliable knowledge graph requires entity and relation extraction (e.g., parsing code, analyzing dependencies), which can be labor-intensive and may fail on dynamically generated code or complex build setups.
- **Prompt size and token budget constraints:** If the retrieved subgraph is large (many related classes, modules, dependencies), summarizing all necessary context into a prompt may exceed the LLM's token limit or reduce clarity.
- **Retrieval and computation cost:** Traversing large graphs, selecting subgraphs, summarizing context, and then running the LLM can be more computationally expensive and slower than simple vector-based RAG, especially for many small queries.
- **Potential noise or irrelevant context:** Overly aggressive graph traversal might bring in too much unrelated code context (e.g., distant dependencies), leading the LLM astray or diluting focus.

Given these trade-offs, GraphRAG is especially well suited for domains like source code test generation, where relational structure matters, but requires developer supervision to maximize benefits while managing costs and complexity.

2.5 Orchestration of LLM agents and LangGraph

As systems grow more complex, researchers have shifted from single-shot prompts toward multi-agent or multi-step pipelines where each agent has a narrow role (analysis, retrieval, generation, validation, etc.). Orchestration frameworks like LangGraph (part of the LangChain ecosystem) make it straightforward to define directed graphs

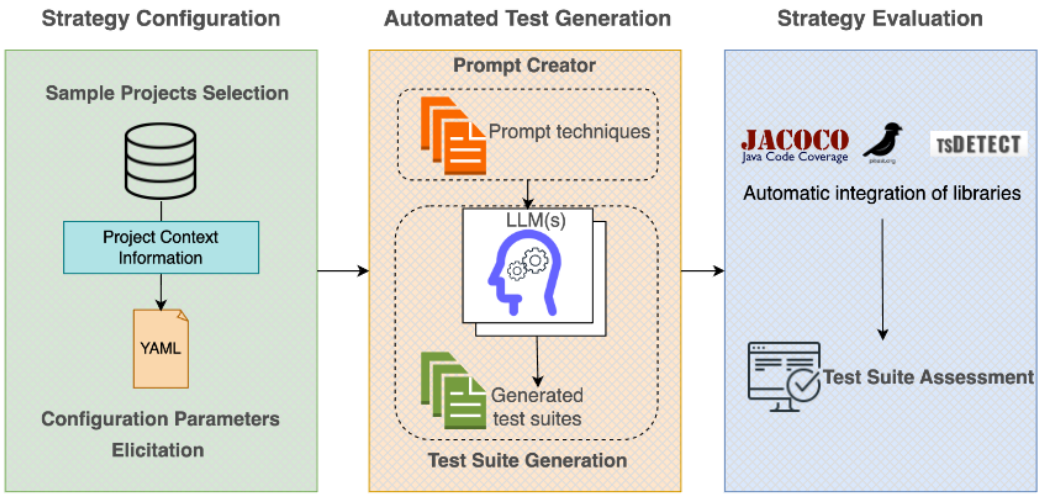


Figure 2.2: Overview of AGONETEST framework[8].

of agents, pass structured state between nodes, implement loops and retries, and separate deterministic logic from generative components. LangGraph's design matches the thesis' need for modular, inspectable workflows that combine LLM agents with deterministic scripts (for example, final test assembly).

2.6 Related systems: AgoneTest, ASTER, and GenUTest

This section briefly summarizes three representative systems and compares their methods, strengths, and limitations relative to the pipeline proposed in this thesis.

2.6.1 AgoneTest

AgoneTest is a recent framework for automated creation and evaluation of unit tests using large language models. It focuses on generating and assessing test suites for Java projects, providing an end-to-end evaluation pipeline that helps compare LLMs and prompting strategies under realistic conditions. AgoneTest emphasizes empirical assessment (how well LLMs do compared to human tests or other baselines) and provides tooling for measuring quality metrics such as compilability, coverage, and flaky behavior. Its strength lies in standardized evaluation and large-scale comparison of models and prompts, but it primarily treats test generation as a (single or few-shot) generation + evaluation task rather than a finely modular, graph-backed multi-agent workflow[8].

2.6.2 ASTER

ASTER (Natural and Multi-language Unit Test Generation with LLMs) presents a pipeline that combines lightweight static analysis with LLM prompting to produce com-

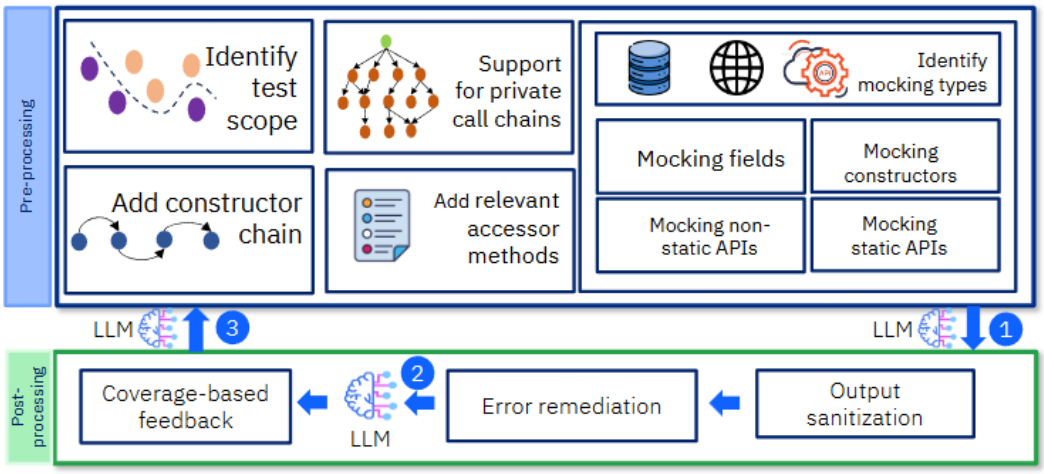


Figure 2.3: Overview of ASTER. 1, 2, 3 represent test-generation, test-repair, and coverage-augmentation prompts[9].

pilable, higher-coverage, and more natural tests across languages (Java and Python). ASTER's preprocessing extracts method-level context and environment requirements (for example, mocking needs), and its postprocessing performs iterative repairs and coverage augmentation. The key idea is that static analysis provides precise, local context to the LLM, which reduces hallucinations and improves executability. ASTER demonstrates strong empirical results showing that analysis guided prompting can outperform some earlier approaches in both coverage and developer-perceived naturalness. Compared to ASTER, the system in this thesis shares the idea of analysis guided generation but expands the retrieval layer from lightweight static context toward graph based retrieval (GraphRAG) and decomposes the workflow into specialized agent nodes coordinated with LangGraph[9].

2.6.3 PolyTest

PolyTest focuses on improving the quality, diversity, and consistency of LLM-generated test suites by leveraging a polyglot generation strategy. Instead of relying on a single prompt–response cycle, PolyTest prompts the LLM to generate tests in multiple programming languages and then unifies these results into a single, coherent test suite. The underlying idea is that different languages elicit different reasoning paths from the model, leading to complementary insights and reducing hallucinated or inconsistent logic. PolyTest also incorporates self-consistency mechanisms to filter or reconcile conflicting tests[7] .

While PolyTest substantially improves the robustness of LLM-generated tests, its core workflow still revolves around direct prompting rather than graph-based retrieval or detailed project-structure analysis. Moreover, PolyTest does not perform iterative validator/repair cycles nor does it perform graph aware context enrichment, which is

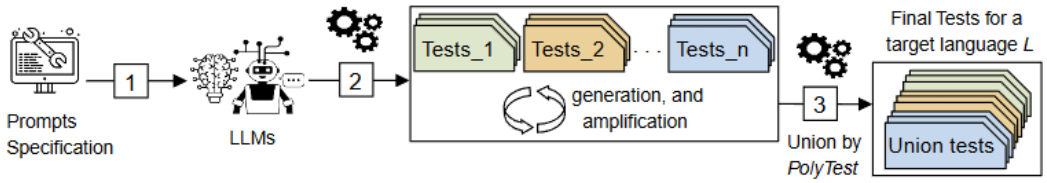


Figure 2.4: Overall approach of PolyTest. It covers two setups: 1) One generation of tests for n languages and 2) n generations for a single language. It also include three steps, generation, amplification, and reduction of tests[7].

a key difference between it and the system proposed in this thesis.

2.6.4 Comparison summary

- **Approach:** AgoneTest emphasizes standardized evaluation of LLM-generated test suites; ASTER focuses on prompting the large language models with the guidennce of static analysis; PolyTest emphasizes polyglot test generation, diversity, and self-consistency. The pipeline proposed in this thesis combines ideas from ASTER (analysis-driven guidance) and from RAG (graph-based relational retrieval), while using a LangGraph multi-agent pipeline that, like AgoneTest, incorporates evaluation and validation as first-class steps.

- **Strengths:** ASTER demonstrates that local static analysis significantly improves test plausibility and coverage; AgoneTest provides rigorous evaluation methodology; PolyTest shows that diversity and multi-language prompting can reduce hallucinations and increase test robustness. The system proposed in this thesis takes inspiration from these strengths by using code summaries + GraphRAG for context retrieval, an iterative validator/rewriter loop to enhance correctness, and a deterministic finalization step for reproducibility and consistency.

- **Limitations and gaps:** AgoneTest and ASTER both improve LLM prompting but neither incorporates graph-based relational retrieval for modeling code-level relationships; PolyTest focuses on diversity across languages, but not on codebase-wide structural understanding or agent-level decomposition. This thesis addresses those gaps by explicitly integrating a graph retrieval layer (GraphRAG) and decomposing responsibilities into specialized LangGraph nodes: summarization, graph construction, scenario generation, test writing, validation, rewriting, caching/saving, and deterministic merging.

2.7 Takeaways for this thesis

The literature shows that

1. combining program analysis with LLM prompting improves test quality (ASTER),
2. standardized evaluation frameworks enable consistent comparison across models and methods (AgoneTest), and
3. diversity and consistency oriented prompting strategies (PolyTest) reduce hallucination and strengthen test robustness.

Building on these insights, this thesis adopts analysis-guided summarization, augments retrieval with a graph-based RAG layer, and organizes agents into a LangGraph pipeline supporting iterative validation, rewriting, and deterministic final assembly.

Chapter 3

System Architecture

This chapter describes the overall architecture of the proposed test-generation system and explains the purpose and functionality of each node in the LangGraph pipeline. The system is designed as a multi-agent workflow, where each agent performs a well-defined task and passes its output to the next agent in the sequence. This structure helps reduce the amount of context each agent needs to handle at once, while still allowing the system to maintain a global understanding of the project under test.

The entire pipeline is implemented in Python and managed using the LangGraph framework, which enables the creation of directed graphs of LLM-powered nodes, making it well-suited for workflows that require iterative steps, branching logic, memory sharing, and stateful agents.

3.1 Overview of the Architecture

The system consists of eight nodes:

1. **summarization_node**
2. **grapher_node**
3. **stub_ideas_node**
4. **test_writer_node**
5. **validator_node**
6. **test_rewriter_node**
7. **test_saver_node**
8. **save_all_tests_node**

Each node operates on a shared state object, which stores these information:

- extracted knowledge about the source code,
- generated test case scenarios,
- previously saved tests,
- feedback from the validation step,
- amount of validations to circumvent infinite loops,
- names of all the classes from the system under test,
- name of the class, for which we are currently generating tests for

The workflow follows a loop-like pattern. After each test is generated, the system validates it, rewrites it if needed, and either continues generating new scenarios or finalizes all tests into a single output file.

The following sections describe each node in detail.

3.2 Summarization Node

The `summarization_node` is the entry point of the pipeline. Its responsibility is to analyze the repository file-by-file and produce structured information about the project, which then can be used to create a knowledge graph later in the pipeline. The format, that the agent in this node is instructed to use for it's output, focuses on three main structures:

1. Entities, where each one can have a:
    - Type, describing what it is: class, field, method or variable,
    - Attributes, representing interesting details about the entity, for example a methods accessor or return type,
    - description, containing context about the entity, for example what the agent thinks the method does.
2. Relationships between the different entities. The declaration of each perceived relationship has to be supported with some evidence,
3. Context, containing information not specified in the structures above, like the name of the file, comments or inferred insights

Instead of sending the entire source code into a single prompt, the system processes files incrementally. For each file, the `summarization_node` generates a summary that is later combined into a single knowledge file. These summaries will also be stored

in a vector database, so that in future steps we can access it with traditional RAG, effectively combining these two proven retrieval methods for the best possible results.

This approach ensures that the agent stays within token limits while still capturing the essential details of the project.

3.3 Grapher Node

The `grapher_node` initializes a local instance of Microsoft's GraphRAG. The purpose of this node is to convert the textual summary produced earlier into a graph representation that captures relationships between code components, such as:

- which classes import or call each other,
- how methods interact,
- how data flows through the system.

For the system developed for this thesis, GraphRAG is configured to run locally using Ollama, allowing the entire pipeline to run on local hardware. GraphRAG builds a knowledge graph from the summarization output and exposes a query interface that later agents use to retrieve context.

This graph-based retrieval is especially usefull for codebases, that are too large or complex for traditional vectorized keyword search.

3.4 Stub Ideas Node

The `stub_ideas_node`'s purpose is to generate descriptions of test case scenarios, that can be used in the input prompt for the `test_writer_node`. It queries the knowledge graph to gather all relevant components for a given target entity, including:

- related classes,
- helper functions,
- expected inputs and outputs,

The `stub_ideas_node` then combines this information with information retrieved using traditional RAG. The agent, with the help of all this information about the system under test, produces a structured list of test case scenarios. These scenarios serve as a blueprint for the actual test code.

In future versions of the system, a Redis database may also be introduced to store the original source code fragments for even more accurate retrieval.

3.5 Test Writer Node

The `test_writer_node` transforms the generated scenarios into real Java test code. This agent receives:

- the test case stub generated by the previous node,
- source code context with the help of traditional RAG,
- global information from the summarization and knowledge graph.

The agent is instructed to produce deterministic, syntactically valid tests using the project's preferred testing frameworks and libraries (usually `junit` and `mockito`). The output is a block of code corresponding to the given scenario.

This node must carefully handle import statements, mock objects, exceptions, and other details that affect the validity and executability of the test.

3.6 Validator Node

Once a test is generated, the `validator_node` analyzes the test's correctness, completeness, and logical consistency. It checks for issues such as:

- incorrect assumptions about the code under test,
- missing assertions,
- logical errors,
- unnecessary or redundant steps,
- potential hallucinations or references to non-existent methods.

The validator does not rewrite the code itself. Instead, it produces structured feedback describing what must be fixed. If the validator approves the test, or the system exceeds the maximum amount of retry attempts, the workflow proceeds to saving the tests. If not, the test will be reviewed and rewritten another time.

3.7 Test Rewriter Node

If the validator rejects a test, the `test_rewriter_node` is triggered. This agent receives:

- the original test,
- the validator's feedback,
- relevant code context.

Its task is to rewrite the test so that it meets all quality criteria. The rewritten test is then returned to the validator for another evaluation. This loop continues until the validator approves the output or the retry limit is reached.

This iterative refinement process helps reduce LLM hallucinations and significantly improves the consistency of the generated tests.

3.8 Test Saver Node

Once a batch of tests pass validation, the `test_saver_node` stores the generated tests in memory. It also clears any temporary shared state so that the next test can be generated with a clean environment.

Based on the remaining tasks, the node decides whether to:

- return control to the `stub_ideas_node` to generate additional test scenarios, or
- proceed to the final test aggregation step.

This decision is made dynamically depending on whether all files have been processed.

3.9 Save All Tests Node

The `save_all_tests_node` is the final step of the pipeline. Unlike the other nodes (except the `test_saver_node`), this step is not powered by an LLM. Instead, it uses a deterministic script to:

- merge all generated tests into a single file (or multiple files, depending on configuration),
- ensure a consistent formatting style,
- write the final output to disk.

This separation ensures reliability: even if LLM-generated tests vary slightly in structure, the final output is produced by predictable, reproducible logic.

3.10 Summary

The architecture is built around the idea that breaking test generation into smaller steps produces more reliable results than relying on a single large model. Each agent has a narrow, clearly defined responsibility, and the system uses both traditional RAG and advanced graph-based retrieval techniques to maintain context. LangGraph provides the management logic needed to make the system robust, iterative, and scalable. It also makes the system modular, which is important for future development and continued improvements.

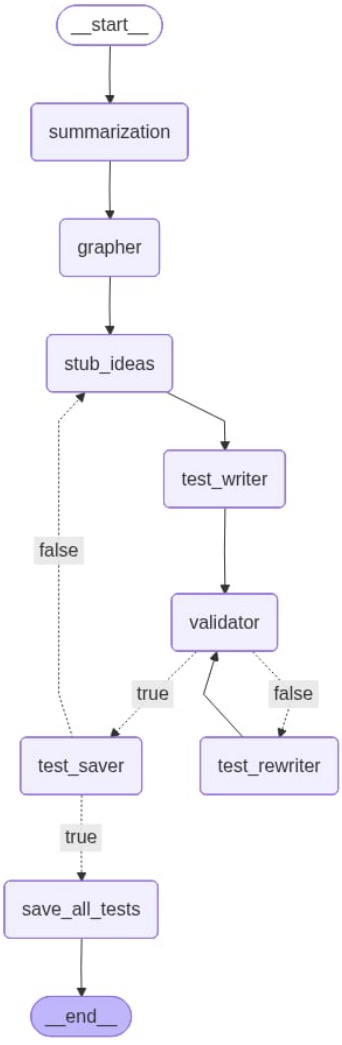


Figure 3.1: Visual representation of the system's architecture

Chapter 4

Evaluation

4.1 Experimental Setup

4.1.1 Hardware and Software Configuration

4.1.2 Models and Tools Used

4.1.3 Selected Codebases

4.2 Evaluation Methodology

4.2.1 Test Quality Metrics

4.2.2 Coverage Measurement

4.2.3 Error and Defect Categorization

4.3 Results

4.3.1 Summarization Node Evaluation

4.3.2 GraphRAG Retrieval Performance

4.3.3 Quality of Test Scenarios

4.3.4 Quality of Generated Tests

4.3.5 Validator and Rewriter Loop Effectiveness

4.4 Comparison with Baselines

4.4.1 Single-Prompt LLM without RAG

4.4.2 Single-Prompt LLM with RAG

4.4.3 Proposed pipeline without RAG

Chapter 5

Future work

5.1 Enhancing Retrieval Mechanisms

5.1.1 Hybrid RAG Approaches

5.1.2 Integration with Vector Databases

5.1.3 Dynamic Graph Updates

5.2 Improving the Multi-Agent Workflow

5.3 Model-Level Enhancements

5.3.1 Fine-Tuning for Code Understanding

5.3.2 Domain-Specific Training Data

5.3.3 Long-Context Model Integration

5.4 Pipeline Extensions

5.4.1 Support for Multiple Programming Languages

5.4.2 Expanding Test Types (Integration, Property-Based, Fuzzing)

5.4.3 Automated Mock and Fixture Generation

5.5 Operational Improvements

5.5.1 Caching and Performance Optimization

5.5.2 Scalability for Large Repositories

5.5.3 Developer Tooling and Integration

Chapter 6

Conclusion

6.1 Summary of Contributions

6.2 Key Findings

6.3 Limitations

6.4 Future Work Directions

6.5 Final Remarks

Bibliography

[1] Marcin Andrzejewski, Nina Dubicka, Jędrzej Podolak, Marek Kowal, and Jakub Siłka. Automated test generation using large language models. *Data*, 10(10):156, 2025.

[2] Gordon Fraser and Andrea Arcuri. 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*, pages 416–419, 2011.

[3] Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yixin Dai, Jiawei Sun, Haofen Wang, and Haofen Wang. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, 2(1), 2023.

[4] Marios Gkikopouli and Batjigdrel Bataa. Empirical comparison between conventional and ai-based automated unit test generation tools in java, 2023.

[5] Haoyu Han, Yu Wang, Harry Shomer, Kai Guo, Jiayuan Ding, Yongjia Lei, Mahantesh Halappanavar, Ryan A Rossi, Subhabrata Mukherjee, Xianfeng Tang, et al. Retrieval-augmented generation with graphs (graphrag). *arXiv preprint arXiv:2501.00309*, 2024.

[6] Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. *arXiv preprint arXiv:2408.02479*, 2024.

[7] Djamel Eddine Khelladi, Charly Reux, and Mathieu Acher. Unify and triumph: Polyglot, diverse, and self-consistent generation of unit tests with llms. *arXiv preprint arXiv:2503.16144*, 2025.

[8] Andrea Lops, Fedelucio Narducci, Azzurra Ragone, Michelantonio Trizio, and Claudio Bartolini. A system for automated unit test generation using large language models and assessment of generated test suites. In *2025 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 29–36. IEEE, 2025.

[9] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Multi-language unit test generation using llms. *arXiv preprint arXiv:2409.03093*, 2024.

[10] Rangeet Pan, Myeongsoo Kim, Rahul Krishna, Raju Pavuluri, and Saurabh Sinha. Aster: Natural and multi-language unit test generation with llms. In *2025 IEEE/ACM 47th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 413–424. IEEE, 2025.

[11] Benny Pasternak, Shmuel Tyszberowicz, and Amiram Yehudai. Genutest: a unit test and mock aspect generation tool. In *Haifa Verification Conference*, pages 252–266. Springer, 2007.

[12] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024.