UNIVERSITY OF TWENTE.

UNIVERSITY OF TWENTE | DESIGN PROJECT

AI Patcher

A Technical Framework for Autonomous Software Repair

AUTHORING TEAM

Thijmen Welberg

Andrey Ivanov

Benjamin Cojocaru

Furqan Saleh

Roman Laduș

Ion Negru

CLIENTS & SUPERVISION

Mattia Napoli

Thijs van Ede

April 17, 2026

Contents

1. Introduction

The project aims to create an AI-powered patching infrastructure specifically for open-source software, with a focus on projects integrated with OSS-Fuzz. OSS-Fuzz is an open-source fuzzing framework developed by Google. Fuzzing involves supplying randomized inputs to applications to identify crashes, bugs, and security vulnerabilities. This approach allows developers, maintainers, and researchers to automatically detect issues in open-source software on a large scale. Currently, most identified vulnerabilities are addressed manually by developers, which can be a time-consuming process. The main goal of this project is to automate this process by utilizing large language models (LLMs) to analyze vulnerability reports, identify the bugs and their root causes, and generate corresponding code patches. The system accepts a vulnerability report ID as input from the user, retrieves the associated report, and extracts relevant information about the issue. This information is then used to prompt a large language model (LLM) to generate a potential patch. The patch is automatically applied to the source code within a local Docker container, where the project is rebuilt and tested against the failing input to verify the fix. A key objective of the system is its modularity, allowing researchers to easily switch between different LLMs to evaluate their effectiveness in automated software repair.

1.1. Project Phases

This project was divided into four phases. Phase one, taking place during weeks 2 and 3, involved creating the project proposal for our client, which included a requirement analysis and the development of a use case diagram to ensure our ideas aligned with the client's wishes. Phase two, spanning weeks 2 through 5, focused on designing the high-level system architecture and modular components, during which we created class and sequence diagrams. Phase three, occurring from weeks 4 through 9, was when we implemented the system, concentrating on the "must-have" and "should-have" requirements. Phase four, executed during weeks 8 and 9, was dedicated to testing and implementing the "could-have" requirements, additionally aiming to optimize the system by addressing the non-functional requirements.

2. Domain Analysis

In this chapter, the identification of the systems domain is discussed. It starts with the background of the domain, then moves on to the client vision, key stakeholders, and the software environment. Complete understanding of the domain allows development to progress more quickly and facilitates future planning.

2.1. Domain Background

Currently, developers using OSS-FUZZ for their projects must download the project, patch vulnerabilities using issue descriptions, and compile it all manually. This manual process makes development less efficient, increases time consumption, and leads to errors.

2.2. Client's Vision

The client requested a pipeline that outlines the patching process. While the use of a large language model (LLM) was included to generate patches for vulnerabilities, the reliability of the AI-generated patches was not a top priority. The primary goal of the pipeline was to automatically analyze the vulnerability report and identify the specific bug that caused the vulnerability. Additionally, the pipeline needed to modify the source code using the patch generated by the LLM and compile the project to verify whether the vulnerability still existed. Furthermore, the client wanted the flexibility to choose which LLM would be used to generate the patch, and the pipeline needed to be modular to allow for future extensibility.

2.3. Stakeholder Analysis

The application has a limited number of stakeholders. It is not designed for a specific user, so some stakeholders may not have a distinct role but are included as they could become users in the future. **Mattia Napoli** had the role of both our client and project supervisor. He introduced the requirements and was responsible for evaluating our team's development process. **Thijs van Ede**, assisted Mattia in both roles. The users of the system may include **University Researchers**, who frequently develop experimental software, and **Developers**

engaged in open-source projects. The suppliers for this application are the **LLM Providers** (they provide the LLM used to fix vulnerabilities), and **CI/CD engineers**, who are responsible for implementing and maintaining automated development pipelines in which the system is integrated.

2.4. Tools and Technology

The following tools and technology stack were used in the development:

- **Python** - an open-source programming language that was used to develop the pipeline. It was chosen due to the availability of libraries and frameworks that were used in this project, as well as simplicity and quick iterations that became possible.
- **UV** - a Python project manager that manages dependencies and environments
- **Langchain** - a Python framework that helped us interact with the LLM in better fashion by providing context, allowing for post-processing of the responses, using various LLM tools, and improving overall maintainability and development experience.
- **Docker** - we used Docker to interact with OSS Fuzz, provide fixed images and run applications inside of it.
- **OSS-Fuzz** - an open-source project developed by Google which uses fuzzing on open-source programs to find vulnerabilities. We will mainly interacted with it to test LLM solutions and run/compile applications.

3. Global Design Process

3.1. Design Methodology

We used the Agile Design methodology for this project. In the span of 2 week sprints, we iteratively implemented the requirements with feedback from the client every week. In addition, scrum meetings allowed for effective communication and progress management. The following roles were assigned to group members to effectively follow this methodology:

- **Contact person** - Thijmen Welberg
  Thijmen was responsible for all communication with the client and overall communication of the team.

- **Organizational coordinator** - Ion Negru
  Ion was responsible for helping the team with the organizational structure of the project, as well as supervising the project meetings.

- **Quality Manager** - Furqan Saleh
  Furqan ensured that all deliverables were of high quality and met all agreed requirements before submission

- **Meeting minutes coordinator** - Andrey Ivanov
  Andrey was responsible for taking minutes of the meetings with the client

- **Scrum master** - Role Rotation
  Since we followed the Agile method, the Scrum master role rotated weekly between all team members. The Scrum master was responsible for coordinating the stand up meetings, keeping track of each member's individual progress, and identifying the challenges they might be facing.

3.2. Requirement Gathering

To conduct our requirements analysis, we held a meeting with the client to better understand the project's core objectives. During this meeting, we translated the identified needs into

functional and non-functional requirements, as well as user stories.

For prioritization, we focused on the requirements essential for delivering the minimum viable product (MVP). Key functionalities include allowing the user to provide an OSS-Fuzz issue ID, checking its existence, retrieving information from the report, generating a prompt to query a large language model (LLM), applying the patch received from the LLM, and finally recompiling and testing the results. These elements were of the highest priority, as they are fundamental to achieving the project's main objectives.

Apart from these main requirements we focused on some which are also vital for making the tool modular and customizable (such as the possibbility to change the LLM) for future extensibility.

3.2.1. Functional Requirements

Must-have:

1. The system must accept OSS-Fuzz issue ID input from the user.
2. The system must be able to recognize broken or invalid inputs (e.g., malformed reports or non-existent container IDs) and notify the user immediately.
3. The system must fetch project name from the given OSS-Fuzz issue ID
4. The system must fetch fuzz target from the given OSS-Fuzz issue ID
5. The system must fetch the reproducer test case from the given OSS-Fuzz issue ID
6. The system must use the fetched information to start a docker container
7. The system must be operable using a Command Line Interface (CLI)
8. The system must identify the buggy file and and the buggy function to be modified
9. The system must generate a prompt containing all necessary details to query an LLM for a code fix.
10. The system must be able to modify the source code (within the docker image) of the OSS-Fuzz project by applying the patch received from the LLM.

11. The system must be able to recompile the project
12. The system must be able to run fuzzing tests to verify the fix.
13. The system must return an action(Success/Failed) status to the user
14. The system must get the sanitizer report locally
15. The system must provide tools to AI for making the pipeline autonomous

Should-have:

1. The system should allow the user to change the LLM
2. The system should have a configuration argument in the CLI
3. The system should be adaptable to local projects or those not built around OSS-Fuzz with minimal changes.
4. The system should implement a feedback loop (e.g., if the compile fails, feed the error back to the LLM and retry X times).

Could-have:

1. The system could count tokens and optimize the query context to avoid hitting model limits.
2. Instead of just "Fixed/Not Fixed," the system should output the specific lines changed (the diff).

Won't-have:

1. The system will not have any login systems or multi-user support.
2. The system will not attempt to fix new bugs introduced by the patch; it only checks if the original bug is gone.

3.2.2. Non-Functional Requirements

Performance:

1. The system shall be able to read and process data and user inputs in under a minute.
2. The system shall be optimized to minimize execution time
3. The system shall minimize the number of queries made to the LLM to reduce costs and latency.

Security:

1. The system shall be free of vulnerabilities which allow prompt injection
2. The system shall have no memory leaks which can be exploited to run arbitrary code

Usability:

1. A graphical user interface for easier usage by non-technical users.
2. The system shall have documentation on how to run the pipeline
3. The system shall have documentation on how to configure the pipeline.

Reliability:

1. The system shall handle build errors within the pipeline gracefully and provide clear, actionable error messages to the user.

Scalability:

1. The system shall be designed to easily add more analysis and tools.

Maintainability:

1. The system architecture shall be designed in modular components with clear interfaces to ensure maintainability.

2. The system shall abstract all Large Language Model (LLM) interactions behind a generic interface.

3. The system shall have a well-documented codebase to facilitate future extensions and maintenance

Portability:

1. The system shall be able to run on any Operating System

3.2.3. Diagrams

To document the final system design after implementation refinements, we provide a complete set of updated UML views, including structural, functional, interaction, process, and behavioral perspectives. Specifically, the class, use case, sequence, activity, and state machine diagrams are used to describe complementary aspects of the AI Patcher architecture and execution logic. These diagrams are included as reference material in Appendix A, where they are presented in full resolution and aligned with the current codebase.

Class Diagram (Structural View) The class diagram (see Appendix A, Figure 1) presents the static architecture of the AI Patcher system and the relationships between its core components. It highlights the orchestration role of `PatchingPipeline`, the container execution layer (`DockerRuntime`), patch generation abstractions (`LLMHandler` and provider variants), crash context extraction (`CodeExtractor` and crash parsing utilities), and verification components (`BaseTestReproducer` / `OSSFuzzTestReproducer`). The diagram also includes data transfer objects (e.g., `IssueReport`, `SourceFileModel`, `TestcaseReproductionModel`) and the optional agent-based correction loop (`PatchAgent`, `PatchTools`) used for iterative remediation.

Activity Diagram (Process Logic) The activity diagram (see Figure 2) models the control flow of the patching process, including branching and iteration logic. It shows the progression through setup, extraction, patch synthesis, application, and verification, as well as failure-handling paths such as patch rejection, apply anomalies, and retry limits. This

view provides a compact representation of operational logic and decision-making in the implemented pipeline.

Sequence Diagram (Runtime Interaction Flow) The sequence diagram (see Figure 3) illustrates the temporal message flow across the main actors and subsystems during one remediation run. It covers issue retrieval/caching, environment preparation, crash reproduction, target-source localization, patch generation by the selected LLM backend, patch application inside the containerized environment, and verification via OSS-Fuzz-style reproduction. The diagram emphasizes call order, decision points, and feedback loops in the end-to-end execution path.

State Machine Diagram (Behavioral States) Finally, the state machine diagram (see Figure 4, Figure 5) captures how key process entities transition between states during execution (e.g., proposal, apply, verify, success/retry/exhausted in agent mode, or compile/test outcomes in verification mode). It formalizes termination conditions and retry semantics, clarifying when execution is considered successful, recoverable, or terminally failed.

Use Case Diagram (Functional Scope) The use case diagram (see Figure 6) describes the system from the user's perspective, focusing on the main interactions supported by the CLI workflow. These interactions include entering an OSS-Fuzz issue identifier, configuring execution options (provider/model/verbosity/agent mode), generating and applying a patch, and verifying remediation. It also captures optional behaviors such as agent mode and local crash input. Together, the use cases define the practical functional boundary of the current implementation.

4. Design Choices

This section describes the key design decisions made during the development of the AI Patcher system, outlining the options considered, the evaluation criteria, and the rationale for the final selections.

4.1. System Architecture

The central architectural decision was to structure the system as a pipeline of independent, loosely-coupled modules rather than a monolithic script. This decision was driven by three primary concerns:

- **Replaceability:** The client explicitly required that different LLMs can be swapped in and out without modifying the surrounding system. This naturally leads to an interface-based design in which the LLM interaction, for instance, is abstracted behind a common `LLMHandler` interface.
- **Testability:** By separating concerns into distinct modules with well-defined interfaces, each component can be tested in isolation using mock implementations, avoiding the need to trigger real Docker containers and LLM API calls during unit tests.
- **Research Extensibility:** The pipeline supports future research directions without requiring a full rewrite. For example, a researcher wanting to compare OSS-Fuzz with another vulnerability database only needs to implement a new `IssueStorage` and `fetch()` function.

All components that have multiple implementations use the **factory function pattern** for instantiation. This isolates the creation logic from the usage logic and allows the CLI to select providers dynamically by name (e.g., `-provider openai`).

4.2. LLM Selection & Patch Generation

The core of the remediation pipeline lies in the selection of an appropriate reasoning engine and the strategy used to apply its suggestions to the codebase. To achieve modularity, the

system abstracts Large Language Model interactions behind a generic interface, allowing for the evaluation of different models based on their effectiveness in software repair. This section details the criteria used to evaluate these models and the technical justification for adopting a specific patch generation approach over more expansive rewriting methods.

For LLM selection, criteria included code understanding, context window size (minimum 16K tokens, 128K+ strongly preferred), instruction follow-up, cost, privacy, and modularity. Google Gemini 2.5 was found to offer the best cost-to-performance ratio for typical prompt sizes with its 1M+ token context window. OpenAI GPT-4o produced the most consistent unified diff format, while Ollama provides a fully local alternative essential for privacy-restricted environments.

When designing the patch generation component, three high-level strategies were evaluated: Whole-File Rewriting, Function-Level Rewriting, and Unified Diff Generation. Whole-File Rewriting was rejected to avoid unnecessary resource usage and the risk of exceeding model token limits. Function-Level Rewriting was also bypassed as it could still introduce regressions or lose critical file-level context required for a verified fix. In the end, **Unified Diff Generation** was chosen because it:

- Produces the smallest possible change, avoiding unintended modifications.
- Is directly applicable via the standard POSIX `patch` command available on all Linux systems.
- Makes the change explicit and easily reviewable as a diff.
- Aligns perfectly with how human developers submit vulnerability fixes in practice.

4.3. Crash Analysis Techniques

For extracting the crash location (file name and line number) from the AddressSanitizer (ASan) output, a **Regex-Based Parsing** approach was chosen over AST/Structured Analysis. This was due to its simplicity, speed of implementation, and robustness to minor format variations across different LLVM versions.

For extracting the specific C/C++ function body from the source file, **Brace-Matching** was chosen over Tree-sitter (AST Parser) because it requires zero additional dependencies (pure Python), works on any C/C++ dialect, and handles non-standard syntax, macros, and compiler extensions gracefully.

4.4. Docker-First Approach

The decision to perform all compilation, patching, and testing inside Docker containers was driven by:

- **Security:** Bounding the theoretical risk of executing LLM-suggested code within an isolated environment.
- **Reproducibility:** Using official OSS-Fuzz Docker images guaranties that the crash reproduced locally is identical to the one on the OSS-Fuzz infrastructure.
- **Platform Independence:** Delegating compilation and testing to a Linux Docker container insulates the pipeline from host OS differences.
- **Isolation from Host State:** Prevents the compilation of large C/C++ projects from interfering with the host system's package manager.

Volume mounts were optimized for performance and safety. The `/out` (fuzzer binaries) and `/work` (build artifacts) directories are mounted to the host to persist compiled binaries, reducing recompilation time from up to 30 minutes to just seconds. However, project source files are *not* mounted to ensure that the host source tree remains unmodified.

4.5. Agent Loop (LangGraph)

The agent mode implements retry explicitly and structurally using LangGraph's state machine, addressing limitations of a stateless loop. The LangGraph agent implements a directed graph where each node is a Python function (`propose_patch`, `apply_patch`, `verify_patch`) and edges are conditional based on the state's status.

At each retry, the agent accumulates a specific error context from the failed attempt (e.g., diff validation errors, `patch` command outputs, or Clang compilation errors). This feedback

is appended to the agent's feedback field for the next LLM query, transforming retries into a guided, self-correcting search.

4.6. Additional Technical Choices

For caching issue reports, a flat JSON file was chosen over SQLite because it requires zero setup, is highly portable, and provides sufficient performance for the expected scale of fewer than 100 entries. The CLI was built using Typer and Questionary to leverage natural type hints integration, rich auto-generated help, and interactive menus, which significantly improve developer ergonomics compared to argparse. Finally, a substantive change validator serves as a patch quality gate, running before patch application to detect whitespace-only or formatting changes and preventing the pipeline from wasting 30–60 seconds of computation on unnecessary recompilation cycles.

5. Implementation & Development

5.1. OSS-Fuzz Integration

5.1.1. Issue Fetching

The primary entry point for issue retrieval is `src/api/sources/oss_fuzz.py`. This module is responsible for converting a raw OSS-Fuzz issue ID into a fully populated `IssueReport` data object. The fetch sequence proceeds as follows:

```
fetch(issue_id: int) -> IssueReport
    |
    ├── Construct URL: https://issues.oss-fuzz.com/issues/{issue_id}
    ├── HTTP GET (requests.get with 30s timeout)
    ├── Check HTTP 200 -> raise ValueError on failure
    ├── extract_report_data(html)      -> raw JavaScript-encoded payload
    ├── extract_detailed_report(html)  -> plain text report block
    └── parse_report_text(issue_id, report_text) -> IssueReport
```

The OSS-Fuzz issue tracker embeds structured vulnerability data in a JavaScript variable named `defrostedResourcesJspb` within the HTML page. The module uses a regular expression to locate and extract this payload. Because the payload is a JavaScript array literal (not pure JSON), the extraction logic isolates the payload string via regular expression and then delegates to a standard JSON parser (json.loads) to handle the nested structures and escaping.

The `extract_detailed_report` function independently locates the human-readable text block labelled "Detailed Report:" in the same HTML page. This block contains the crash type, sanitizer name, fuzzing engine, crash state lines, and links to the reproducer test case. It is parsed by `parse_report_text` using a series of targeted regular expressions.

Pattern Target	Regex Fragment
Project name	`Project:\s+(.+)`
Fuzzing engine	`Fuzzing Engine:\s+(.+)`
Fuzz target	`Fuzz Target:\s+(.+)`
Job type	`Job Type:\s+(.+)`
Platform	`Platform Id:\s+(.+)`
Crash type	`Crash Type:\s+(.+)`
Crash address	`Crash Address:\s+(.+)`
Sanitizer	`Sanitizer:\s+(.+)`
Testcase ID	`testcase_id=(\d+)`
Testcase download URL	`https://oss-fuzz.com/testcase.*`

Table 1: Key regex patterns used in parsing.

5.1.2. HTML Parsing & Data Extraction

The OSS-Fuzz web interface does not expose a formal REST API, so the module must scrape HTML. This design decision was chosen over unofficial API endpoints because it is more stable against authentication changes. The HTML parsing strategy relies on two independent mechanisms:

Mechanism 1 — JavaScript Payload Extraction:

The OSS-Fuzz HTML page includes a block resembling:

```
<script>
  var defrostedResourcesJspb = [["ClusterFuzz", ["42515068", "libpng", ...]]]
</script>
```

A regular expression matches `defrostedResourcesJspb\s*=\s*(\[.*?\]);` with `re.DOTALL` to span multiple lines. The inner content is then decoded to extract the list of field values in positional order.

Mechanism 2 — Detailed Report Block:

A second regex isolates the "Detailed Report:" text section:

```
Detailed Report:
================
Crash Type: Heap-buffer-overflow
Crash Address: 0x602000003d51
```

```
Crash State:
  png_read_row
  png_read_rows
  ...
```

Both extraction mechanisms are independently tested to ensure the pipeline degrades gracefully when one source is unavailable (e.g., access-controlled issues).

5.1.3. IssueReport Data Model

The `IssueReport` class (defined in `src/api/dto.py`) uses Pydantic v2 for validation. All fields are validated at construction time; invalid HTTP URLs, missing required fields, or type mismatches raise a `ValidationError` before reaching the pipeline.

5.1.4. Issue Caching

To avoid repeated HTTP requests during development and iterative testing, all fetched issues are persisted in a local JSON file. The storage layer is implemented through the `JSONIssueStorage` class (`src/api/storage/variants/json_files.py`), which implements the `IssueStorage` protocol. The default storage path is `./issues.json`. The file format is a JSON array of serialised `IssueReport` objects. The pipeline's `fetch_issue` static method checks the cache first. If the issue is already cached and `-refresh` has not been specified, the cached version is returned without making an HTTP request.

5.2. Docker Runtime

5.2.1. Container Lifecycle

The `DockerRuntime` class (`src/runtimes/docker.py`) manages the full lifecycle of the Docker container used for building, patching, and testing. It wraps the official Docker SDK for Python.

```
User calls pipeline.run()
        v
DockerRuntime.start(image_name, project_name, out_host, work_host)
        |
        ├── [image not present locally]
        |       └── pull_with_progress(image_name)
        |
        ├── [container named "ossfuzz-{project}" already exists]
        |       └── Reuse existing container
        |
        ├── [new container needed]
        |       ├── client.containers.create(
        |       |       image=image_name,
        |       |       name="ossfuzz-{project}",
        |       |       command=["sleep", "infinity"],
        |       |       volumes={out_host: "/out", work_host: "/work"},
        |       |       mem_limit="8g",
        |       |       memswap_limit="16g",
        |       |       tty=True, detach=True
        |       |   )
        |       └── container.start()
        |
        v
    Container is running; ready for commands
        |
        ├── run_command(cmd) ... (repeated calls for compile/reproduce)
        ├── read_file(path)  ... (reads source files from container FS)
        └── write_file(path, content) ... (writes patch files)
        |
        v
```

```
DockerRuntime.stop(image_name)
        ├── container.stop()
        └── container.remove()
```

The container is given an infinite sleep command as its entrypoint (`sleep infinity`). This keeps the container alive between sequential `docker exec` calls, which is essential because each `run_command` invocation opens a fresh exec session.

5.2.2. DockerRuntime API & File I/O

Every command is executed as: `docker exec -it {container_name} bash -c "{cmd}"`. The method collects both stdout and stderr via the Docker SDK's exec API and returns them as a single string.

Docker does not expose a simple file-read API; files must be transferred using Docker's archive (tar) API. **Reading a file (`read_file`):**

```
def read_file(self, file_path: str) -> str:
    try:
        bits, _ = container.get_archive(file_path)
        buf = BytesIO()
        for chunk in bits:
            buf.write(chunk)
        buf.seek(0)
        with tarfile.open(fileobj=buf) as tar:
            member = tar.getmembers()[0]
            f = tar.extractfile(member)
            return f.read().decode("utf-8")
    except docker.errors.NotFound:
        ...
```

If the specified path does not exist, `read_file` invokes a multi-stage fallback search including keyword-based heuristics (e.g., if the function contains "uncompr", look for files named `uncompr.c`).

Writing a file (`write_file`):

```
def write_file(self, file_path: str, content: str) -> None:
    buf = BytesIO()
    with tarfile.open(fileobj=buf, mode="w") as tar:
        encoded = content.encode("utf-8")
        info = tarfile.TarInfo(name=os.path.basename(file_path))
        info.size = len(encoded)
        tar.addfile(info, BytesIO(encoded))
    buf.seek(0)
    container.put_archive(os.path.dirname(file_path), buf)
```

Writing files follows a similar approach to reading, using Docker's archive API. First the file content is packaged into a tar archive in memory. This archive is then transferred to the container via the `put_archive` method, which extracts the file at the target path.

5.2.3. Environment Configuration

When compiling and running fuzz targets, the Docker runtime injects a set of environment variables that configure the sanitizer and build flags, such as `SANITIZER` (`address`, `memory`, or `undefined`), `FUZZING_ENGINE`, and corresponding compiler flags.

5.3. Crash Analysis

5.3.1. Stack Frame Parsing

After reproducing the crash inside the container, the pipeline captures the full sanitiser output, typically several hundred lines of AddressSanitizer (ASan) trace. The module `src/test_reproducers/crash_parser.py` is responsible for extracting a single actionable crash location (file, line, function) from this output.

Primary Function: `parse_crash_output(text, project_name=None) -> Optional[ParsedCrash]`

The function processes the raw crash output through the following stages:

Stage 1 - Text Normalisation:

ASan output sometimes wraps long lines; consecutive lines that appear to be continuation of a single frame are merged. A line starting with `#` followed by a digit is treated as a new frame; other indented lines are joined to the preceding frame.

Stage 2 - Frame Extraction:

Each stack frame in ASan output has the format:

```
#4 0x7f3a1b2c3d4e in png_read_row /src/libpng/pngrutil.c:1234:56
```

The regular expression used to match frames captures the frame index, function name, absolute file path, and line number.

Stage 3 - Frame Filtering:

Frames from the toolchain infrastructure are excluded by checking the file path against a blocklist of path prefixes:

Excluded Path Prefix	**Reason**
`/llvm-project/`	LLVM compiler internals
`/compiler-rt/`	AddressSanitizer runtime
`FuzzerLoop.cpp`	libFuzzer harness
`/libc`	Standard C library
`/usr/`	System libraries
`sanitizer_common`	Sanitizer support code

Table 2: Excluded paths during stack frame parsing.

Stage 4 - Best Frame Selection:

From the remaining project frames, the function selects the shallowest (lowest frame index) that satisfies the following preference order:

1. Frame in `/src/{project_name}/` — most specific to the target project.
2. Frame in `/src/` — within the OSS-Fuzz source directory.
3. Any remaining frame with a valid file path.

5.3.2. Crash State Fallback

When `parse_crash_output` fails to find any suitable frame, the pipeline falls back to `parse_crash_state(crash_state, project_name)`. The `crash_state` input is the list of

function name strings stored in the `IssueReport`. This fallback performs exact file mapping and keyword heuristics to locate a source file, returning the first resolvable frame with line number set to 0 (indicating file-level granularity). This fallback is intentionally lossy to keep the pipeline moving when perfect information is unavailable.

5.3.3. Source File & Function Extraction

Once the crash location is known, the pipeline reads the source file from the container and stores it in a `SourceFileModel` DTO. When the crash parser identifies a specific function name and the source file is available, the pipeline attempts to extract just the body of the crashing function using a brace-matching algorithm with string awareness. This reduces prompt size and focuses the model's attention on the relevant code. The brace-matching approach handles the vast majority of real-world functions without any dependency on a native compiler.

Raw Name from ASan	**Generated Candidates**
`png_read_row`	`["png_read_row"]`
`LibRaw::copy_bayer(unsigned short*, ...)`	`["LibRaw::copy_bayer", "copy_bayer"]`
`Foo::bar const`	`["Foo::bar", "bar"]`
`operator new(unsigned long)`	`["operator new"]`

Table 3: Candidate name generation for function extraction.

5.4. LLM Integration

5.4.1. Abstract LLM Handler Interface

All LLM interactions are mediated through the abstract `LLMHandler` class defined in `src/llm_handlers/base.py`. This class defines a single public method `generate_patch` that all concrete providers must implement, returning a raw response string containing the patch.

5.4.2. Provider Implementations

The system supports five LLM providers, selectable at runtime via CLI or interactive menu. Provider selection is centralised via a factory function `get_llm_handler` in

`src/llm_handlers/variants/__init__.py`.

Provider	Class	Module	Default Model
OpenAI	`OpenAIHandler`	`variants/openai_.py`	`gpt-4o`
Google	`GoogleHandler`	`variants/google.py`	`gemini-2.5-flash`
Ollama	`OllamaHandler`	`variants/ollama.py`	`llama3:latest`
Perplexity	`PerplexityHandler`	`variants/perplexity.py`	`sonar`
Dummy	`DummyHandler`	`variants/dummy.py`	*(fixed testing response)*

Table 4: Supported LLM Providers.

5.4.3. Prompt Construction Strategy

The prompt assembled is the most critical artefact in the entire pipeline. Its structure directly determines the quality and correctness of the generated patches. The constraint against introducing new identifiers was added after observing that early LLM responses frequently introduced helper variables or called functions that did not exist in the local scope, causing compilation failures. The ascending hunk order constraint was added to avoid POSIX `patch` command failures.

5.4.4. Patch Extraction & Sanitization

The LLM response is a free-text string. The extractor tries two strategies in order:

1. **Markdown Fenced Block:** Searches for a diff wrapped in a markdown code block (“‘`diff ...` “‘).
2. **Raw Diff Detection:** Scans for a line beginning with `---`, `+++`, or `@@`.

Even after extraction, LLM-generated diffs frequently contain subtle formatting errors. The sanitizer applies corrections such as prepending missing leading spaces, recalculating incorrect hunk counts, stripping embedded line numbers, truncating trailing commentary, and normalising wrong file paths in headers.

5.4.5. Substantive Change Validation

Before applying a patch, the pipeline checks whether it contains a substantive change. If the patch is deemed non-substantive (e.g., only whitespace or comment differences), the pipeline

skips application and immediately retries the LLM query with a note in the feedback prompt explaining that the previous response was a formatting-only change.

5.5. Patch Application Pipeline

5.5.1. 4-Phase Orchestration

The `PatchingPipeline` class (`src/pipelines/engine.py`) is the central orchestrator. Its `run()` method sequences four phases and manages the overall execution state. Each phase returns either a success indicator and its output data, or a `TestcaseReproductionModel` with an error stage that short-circuits subsequent phases.

Phase 1 - Environment Setup:

The environment setup phase is the most time-consuming part of the pipeline, often taking 5–30 minutes on first run due to Docker image download and compilation. On subsequent runs, the compiled binary is cached in the `/out` mount and this phase completes in under a minute.

Phase 2 - Targeted Source Extraction:

This phase involves reproducing the crash, parsing the stack trace (or falling back to the crash state), and reading the source file from the container. The brace-matching algorithm is then used to optionally extract the specific function body.

Phase 3 - Patch Generation & Application (Non-Agent Mode):

In non-agent mode, the pipeline implements a simple retry loop with up to n attempts. It queries the LLM, extracts and sanitises the unified diff, checks for a substantive fix, applies the patch using the POSIX `patch` command, and finally verifies that the file was actually changed.

Phase 4 - Verification:

The pipeline forces a recompile using the OSS-Fuzz build script and replays the original crashing test case to verify the fix.

5.5.2. Retry Logic & Feedback Loops

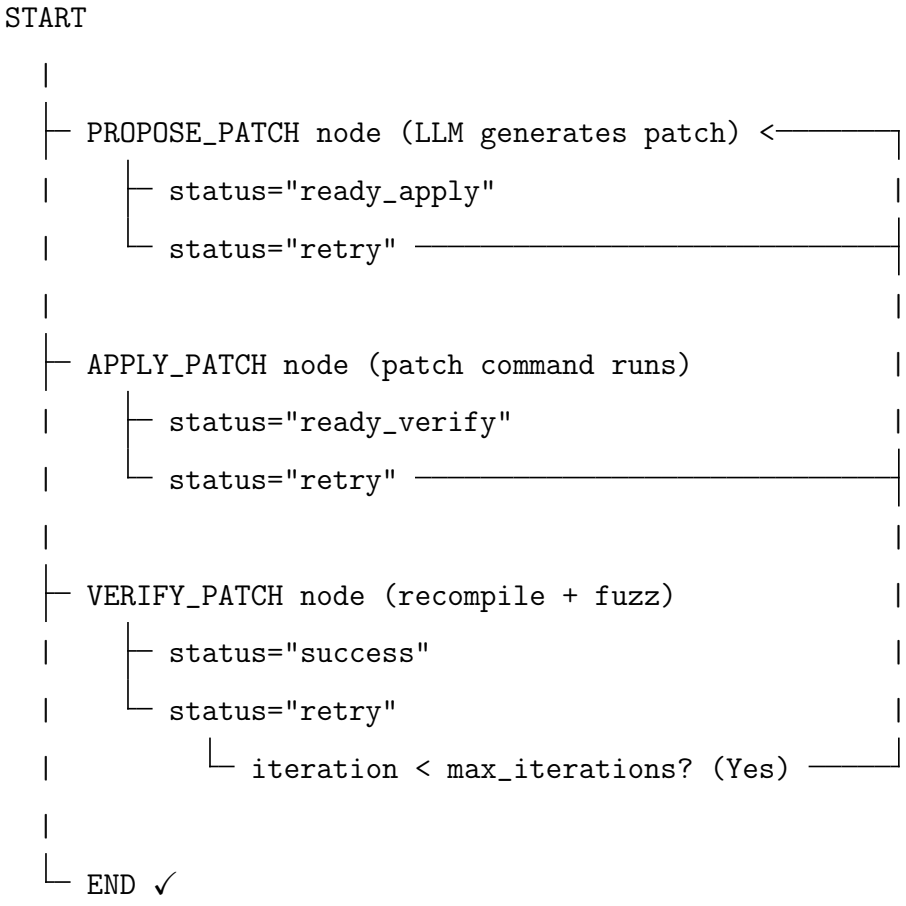
The feedback mechanism distinguishes AI Patcher from a naive one-shot LLM querying system. After each failed attempt, the error output is appended to the next LLM prompt under a "PREVIOUS ATTEMPT FEEDBACK" section. This enables the LLM to self-correct format errors, scope errors, or logical errors based on real compilation and patching feedback.

Failure Stage	**Feedback Content Sent to LLM**
Diff extraction fail	"Response did not contain a valid unified diff block."
Non-substantive patch	"Patch only differs in whitespace/comments — no semantic change."
Patch command fail	Full output of `patch` command including error lines
File unchanged	"Patch applied cleanly but file content is identical to original."
Compilation error	Full compiler error output (clang error messages)
Crash still present	"Recompiled binary still crashes on the test case."

Table 5: Feedback categories sent to the LLM during retry loops.

5.5.3. Agent Mode (LangGraph State Machine)

When invoked with `-agent`, the pipeline delegates Phase 3 to a stateful agent implemented using LangGraph (`src/agents/patch_agent.py`). LangGraph allows the patch iteration loop to be expressed as a directed graph of nodes and conditional edges, rather than an imperative retry loop.



5.6. CLI Implementation

5.6.1. Interactive Mode

The interactive mode (`run_interactive()` in `src/__main__.py`) provides a guided step-by-step interface using the `questionary` library for styled prompts and the `rich` library for formatted output. API key input is handled by a custom function that replaces typed characters with bullets to avoid exposing keys in terminal scrollback history.

5.6.2. Automation Mode

The automation mode provides full scripting capability via the `patch` subcommand (e.g., `python -m src patch -i 42515068 -p google -m gemini-2.5-flash`). Options include specifying the LLM provider, API key, model, toggling the agent loop, and controlling verbosity.

5.6.3. Verbosity & Logging

Three verbosity levels control the volume of terminal output:

1. **Quiet:** Single-line final status (FIXED / FAILED / ERROR) with exit code.
2. **Normal:** Phase headers, key events, timing summary per phase, and total duration.
3. **Full:** All of the above plus LLM prompts, raw responses, patch diffs, and compile output.

When the environment variable `AI_PATCHER_DEBUG_SESSION_ID` is set, the pipeline writes structured JSON logs to `./logs/sessions/debug-{sessionId}.log`, enabling post-hoc analysis of pipeline behaviour across multiple runs.

5.7. Storage & Configuration

5.7.1. JSON Issue Storage

The `JSONIssueStorage` class implements the `IssueStorage` protocol using a simple JSON file (`issues.json`) as its backend. The design deliberately avoids a database dependency, keeping the tool self-contained and portable. Operations like `get`, `upsert`, and `delete` perform linear scans and full file rewrites, which is acceptable for the scale of this research tool (typically fewer than 100 cached issues).

5.7.2. Environment Variables & Logging

API keys for OpenAI, Google Gemini, and Perplexity can be provided either through environment variables (e.g., `OPENAI_API_KEY`) or directly via the `-api-key` CLI flag. The system also supports a local Ollama server configuration.

When the environment variable `AI_PATCHER_DEBUG_SESSION_ID` is set, the pipeline maintains a debug log for each run, writing structured JSON logs to `./logs/sessions/debug-{sessionId}.log`. Each log entry includes a timestamp, event type, and structured payload, enabling post-hoc analysis of pipeline behaviour across multiple runs.

5.8. Verification & Reproducibility

5.8.1. OSSFuzzTestReproducer

The `OSSFuzzTestReproducer` class (`src/test_reproducers/oss_fuzz.py`) is responsible for both the initial crash reproduction (Phase 2) and the post-patch verification (Phase 4). The reproduction process follows these steps:

1. **Fetch testcase:** Downloads the testcase into the container.
2. **Check binary:** Verifies if the compiled fuzzer exists in `/out/`.
3. **Compile:** Runs the OSS-Fuzz build scripts if necessary.
4. **Execute:** Runs the fuzz target against the testcase and captures the output.

5.8.2. Success Criteria

The `is_success` field of `TestcaseReproductionModel` is `True` when the output contains AddressSanitizer, MemorySanitizer, or UBSan crash markers. Note the semantics inversion: in the context of *verifying a patch*, `is_success=False` (crash is gone) is the desired outcome.

The pipeline applies a three-condition check to declare a fix successful:

1. Reached the test stage (no compile errors).
2. No crash markers in output (crash is gone).
3. Fuzz target exited cleanly (exit code 0).

5.8.3. Sanitizer Configuration

The sanitizer is determined by the `IssueReport` and affects compile flags, runtime environment variables, and the Docker image tag used. Supported sanitizers include AddressSanitizer (ASan), MemorySanitizer (MSan), and UndefinedBehaviorSanitizer (UBSan).

5.9. Integration

5.9.1. End-to-End Sequence

The end-to-end data flow begins with the User providing an OSS-Fuzz Issue ID to the CLI. The Pipeline orchestrates fetching the `IssueReport` via the API layer, caching it, and starting the Docker container. The fuzzer is built and the crash is reproduced to extract the target source code and function. The context is sent to the LLM to generate a patch. The pipeline extracts, sanitises, and validates the patch before applying it inside the container. Finally, the target is recompiled and the testcase is replayed to verify the fix, returning the final status to the user.

5.10. MoSCoW Delivery Summary

MVP was achieved by sprint 3, as all Must-have requirements (15) were fully implemented by the end of it. Beyond these essentials, the team completed 75% of the Should-have requirements (the rest were removed from the scope of the project) and successfully delivered 50% of the Could-have features. In addition to this, the total completion of functional requirements is 92%.

6. Testing

Testing an automated vulnerability remediation pipeline presents unique challenges that go beyond conventional software testing. The AI Patcher's core logic depends heavily on non-deterministic external systems, including Large Language Models (LLMs), Docker containers, and the OSS-Fuzz issue tracker. Furthermore, "correct" output in this context is not strictly binary; a generated patch may be syntactically valid and compile successfully, yet fail to fix the underlying vulnerability. Lastly, end-to-end (E2E) execution takes tens of minutes per test case due to Docker image pulls and C++ compilation, making exhaustive automated E2E testing impractical. These constraints fundamentally shaped the project's testing methodology.

6.1. Test Plan

To address the aforementioned challenges, the AI Patcher testing strategy follows a rigid, three-level Test Pyramid approach:

- **Level 1 - Unit Tests:** Every module is tested in complete isolation. All external dependencies (Docker, LLM APIs, HTTP) are mocked. Unit tests must run in under 10 seconds in total, with no network access and no Docker daemon required, enabling them to run seamlessly in any CI/CD pipeline.
- **Level 2 - System/Integration Tests:** These tests evaluate the interaction between two or more modules using realistic mock objects or lightweight environments (e.g., a minimal Docker container that behaves like a real one, with in-memory file storage).
- **Level 3 - User/End-to-End Tests:** The complete pipeline is run against real OSS-Fuzz issues using real Docker containers and live LLM API calls. Due to cost and time, these are executed manually and recorded in a benchmark spreadsheet.

Core Testing Principles:

1. *No live API calls in automated tests:* All LLM and HTTP interactions are mocked to ensure tests remain deterministic, cost-free, and fast.

2. *No Docker daemon requirement in unit tests:* Docker interactions are mocked using the standard `unittest.mock` library. Tests actually requiring the Docker SDK are marked with `@pytest.mark.integration` and excluded from default runs.

3. *Behavioural Testing:* Tests verify the observable output (return values, raised exceptions, state changes) rather than internal implementation details.

The testing framework utilizes `pytest` ($\geq$ 8.3.5) as the primary runner, `pytest-cov` ($\geq$ 6.0.0) for coverage reporting, `pydantic` for DTO fixture construction, and `tarfile` with `io.BytesIO` to simulate Docker file I/O entirely in memory. The project targets an overall code coverage of $\geq$ 69%, with critical parsing modules (e.g., `crash_parser.py`) strictly targeting $\geq$ 95% coverage.

6.1.1. Unit Testing

The unit test suite consists of approximately 201 tests organized in `tests/unit/`, covering all deterministic logic, parsing algorithms, and state management.

CLI & Success Semantics Tests:
The CLI success detection logic (`_is_pipeline_success`) is a critical function with precise semantic requirements. The test suite verifies all combinations of the three conditions that determine success: reaching the test stage, eliminating the crash, and returning a clean exit code. For instance, if the pipeline reaches the "test" stage, yields an exit code of 0, and returns no crash, it is a success. If it compiles but the crash is still present (exit code 1), it correctly registers as a failure.

Code Extraction & Signature Tests:
The `CodeExtractor` module is extensively tested against complex C++ constructs. Tests cover the candidate name generation logic, which demangles ASan-reported function names (e.g., converting `LibRaw::copy_bayer(unsigned short*)` into searchable patterns like `["LibRaw::copy_bayer", "copy_bayer"]`). Furthermore, a custom brace-matching algorithm is tested against edge cases such as deeply nested blocks, string literals containing brace characters (`"{"`), and block comments (`/* { */`).

```
def test_string_with_braces():
```

```
    source = '''
int foo(int x) {
    const char* fmt = "value = { %d }";
    if (x > 0) { return 1; }
    return 0;
}
int bar() { return 42; }
'''
    extractor = CodeExtractor()
    result = extractor.extract_function(source, "foo")
    assert "const char* fmt" in result
    assert "bar" not in result
    assert result.count('{') == result.count('}')
```

Crash Parser Tests:

The crash parser tests rely on recorded, real-world AddressSanitizer (ASan), MemorySanitizer (MSan), and UndefinedBehaviorSanitizer (UBSan) outputs rather than generated strings. The test suite validates that the parser correctly filters out LLVM compiler frames and fuzzer drivers (e.g., `FuzzerLoop.cpp`), selecting the highest-priority frame belonging directly to the targeted project. It also accounts for fallback paths, mapping function names to plausible file names when stack traces are mangled or missing.

Docker Runtime Mocking Tests:

Docker runtime unit tests intercept the `docker.from_env()` call to return a `MagicMock` client. This simulates container behaviour entirely using in-memory data structures. File reading operations via Docker's `get_archive` are tested by generating synthetic tar archives on the fly, allowing the pipeline to test code injection logic without writing to an actual filesystem.

```
def test_read_file_via_tar(mock_docker_client):
    mock_client, mock_container = mock_docker_client
    buf = BytesIO()
    with tarfile.open(fileobj=buf, mode='w') as tar:
```

```
        content = b"int main() { return 0; }\n"
        info = tarfile.TarInfo(name="main.c")
        info.size = len(content)
        tar.addfile(info, BytesIO(content))
    buf.seek(0)

    mock_container.get_archive.return_value = (
        [buf.read()], {"size": len(content)}
    )

    runtime = DockerRuntime()
    runtime._container = mock_container
    result = runtime.read_file("/src/project/main.c")
    assert result == "int main() { return 0; }\n"
```

Pipeline & LLM Handler Tests:

The orchestration engine is tested by passing fully mocked dependencies (Docker, LLM, Reproducer, and HTTP) into the `PatchingPipeline` class. This isolates phase management, testing retry logic triggered when the LLM returns non-substantive or whitespace-only diffs (retrying up to 5 times). For LLM Handlers (OpenAI, Google, Ollama), testing replaces `langchain` module imports at the `sys.modules` level to intercept `invoke()` calls, guaranteeing no API keys are required during the test suite execution.

6.2. System Testing

System testing evaluates the interactions between the AI Patcher's internal modules and its immediate external boundaries, ensuring that the components function correctly when combined.

Docker Integration:

These tests interact with a real Docker daemon but instantiate a minimal container (e.g.,

`ubuntu:latest`) to reduce overhead instead of pulling multi-gigabyte OSS-Fuzz images. The system verifies that container creation with volume mounts succeeds, command execution (`run_command`) accurately captures both exit codes and stdout/stderr streams, and file I/O operations (`write_file` followed by `read_file`) round-trip seamlessly.

LLM Interface Integration:

Live LLM API calls are expensive and inherently non-deterministic, so system testing here utilizes a two-tier approach. First, the LangChain `invoke()` method is mocked at the module level to return pre-recorded responses. This validates that the system correctly constructs the context prompts, parses the raw string response, and extracts the structured JSON/diff outputs. Second, a constrained set of live API tests are executed manually using real API keys to verify network configurations and provider-specific quirks.

Build System & Agent Integration:

We verify that the compilation commands dispatched to the Docker containers are correctly formatted for the OSS-Fuzz infrastructure, including injecting the correct sanitizer flags (ASan, MSan, UBSan). The `PatchAgent` state machine is tested to ensure it correctly loops through the `propose` → `apply` → `verify` states, successfully triggers retries upon compilation failures, and respects the `max_steps` iteration caps.

6.3. User Testing

A round of User Testing was also conduted to evaluate how users react to the platform and how easy the CLI is to use. We were particularly interested in how understandable the system is for new users, including how easily they grasp the system's capabilities and customize it through the CLI. To collect feedback, we took notes on participants' comments during the session and asked each participant to complete a survey. This feedback was used to enhance CLI accessibility and usability.

From a technical perspective, this phase also served as our End-to-End (E2E) testing strategy. Users ran the complete pipeline against a curated benchmark dataset of real OSS-Fuzz issues (representing varying vulnerability classes like Heap-buffer-overflows and Null-dereferences). A patch was only considered successful if the target recompiled, the target process exited cleanly (`exit_code=0`), and the crash was entirely eliminated.

6.4. Test Results

The automated testing suite yielded highly stable results for the core logic. Out of 201 automated unit and system tests, 199 passed successfully, while 2 system tests were intentionally skipped during standard CI runs because they require a live Docker daemon.

Code coverage was measured using `pytest-cov` on the `src/` directory. The overall coverage achieved was approximately 69% across 2,287 statements. Critical components exceeded standard thresholds: `crash_parser.py` achieved 94% coverage, `code_extractor.py` achieved 91%, and the LangGraph `patch_agent.py` reached 85%.

Table 6: Unit Test Coverage Targets & Achievements

Module Group	**Target**	**Rationale**
`crash_parser.py`	$\geq 95\%$	Core parsing logic; high correctness requirement
`code_extractor.py`	$\geq 90\%$	Core extraction logic; brace-matcher needs thorough testing
`api/sources/oss_fuzz.py`	$\geq 85\%$	HTML parsing with numerous edge cases
`pipelines/engine.py`	$\geq 80\%$	Complex orchestration; most paths must be tested
`runtimes/docker.py`	$\geq 80\%$	Docker interactions with multiple fallback paths
`agents/patch_agent.py`	$\geq 85\%$	State machine with multiple transition paths

6.4.1. User Testing

We received a considerable amount of feedback through user testing on how to make the interface more user friendly. Moreover, we were also able to identify more bugs and edge cases in the pipeline. The user testing was done by four researchers at the University along with the client. All users were asked to run the pipeline and configure the different options to patch vulnerabilities from multiple OSS-Fuzz projects. As a result, we were able to identify the following problems and fixed some of them to improve the framework:

1. **Problem**: Users were confused on how to setup and run the pipeline.

   Solution: Improved documentation and README to make the instructions clearer.

2. **Problem**: Users were trying to find the patch applied by the LLM.
   Solution: Show the applied unified diff patch directly in the terminal during the execution of the pipeline.

3. **Problem**: If the API key was entered incorrectly, it was not clearly reported and the pipeline was still executed.
   Solution: Implemented early validation of API keys during the initialization phase, cleanly exiting with a specific authentication error before allocating Docker resources.

4. **Problem**: The Docker logs were too verbose and users missed other important logs because of it.
   Solution: Introduced specific logging verbosity levels (`-quiet`, `-normal`, `-verbose`). Raw Docker build logs are now suppressed by default unless the verbose flag is provided.

5. **Problem**: The pipeline was being executed for projects with no crashes.
   Solution: Added a pre-execution verification step that parses the OSS-Fuzz report and checks the issue status, halting the pipeline if no reproducible crash state is found.

6. **Problem**: API keys must be repeatedly entered.
   Solution: Integrated `python-dotenv` to allow the system to automatically load API keys securely from a local `.env` file or system environment variables.

7. **Problem**: Token usage was not visible.
   Solution: Hooked into LangChain's token tracking callbacks to aggregate prompt and completion tokens, displaying a final cost and usage summary at the end of the run.

8. **Problem**: CLI interface was too complex for general users.
   Solution: Refactored the command-line arguments to use sensible default values for LLM temperature, maximum iterations, and target models, reducing the required arguments to just the issue ID.

9. **Problem**: No clear summary of changes made by the system.
   Solution: Implemented a final execution summary table that outputs the total

execution time per phase, the outcome of the compilation, and the path to the stored patch artifacts.

10. **Problem**: Model configuration options are not visible or validated.

    Solution: Added a `-list-models` flag and strict input validation using Pydantic, instantly rejecting unsupported provider strings and showing users the valid options (e.g., OpenAI, Google, Ollama).

6.5. Evaluation

Testing the AI Patcher pipeline provided vital insights into evaluating non-deterministic and system-dependent software. The most significant takeaway was the necessity of robust mocking boundaries. Migrating to strict `unittest.mock` patching for LangChain and simulating Docker archives entirely in-memory drastically improved test reliability and execution speed. Furthermore, handling LLM non-determinism required architectural adjustments. By implementing a `DummyHandler` that consistently returns a hardcoded, syntactically valid patch, the team isolated the core pipeline logic. This allowed developers to rigorously test the Docker compilation, crash parsing, and agent retry loops without incurring latency or financial costs from real LLM APIs. Future testing work will focus on scaling the user-testing benchmark dataset to over 50 real-world OSS-Fuzz issues and establishing a multi-provider framework to quantitatively compare the accuracy, token efficiency, and syntax correctness of various LLMs against human-written patches.

7. Future Planning

While the current implementation of the automated AI patcher successfully integrates automated crash reproduction and multi-model patch generation within a containerized pipeline, there are still several areas which could be expanded. This chapter outlines the future development roadmap for the project, categorized using the MoSCoW prioritization framework into capabilities the system could implement, methodologies it should refine, and concrete next steps for infrastructure scaling.

7.1. Extended Functional Capabilities (Could)

Multi-File Patching and Higher-Order Reasoning

Currently, the system focuses on localizing and fixing bugs within a limited, single-file scope. Future iterations could implement higher-order reasoning via multi-agent collaboration to handle complex vulnerabilities that require simultaneous changes across multiple files or architectural modules. By employing an orchestrator agent to map the Abstract Syntax Tree (AST) of the wider codebase, the patcher could track variable scopes and cross-module dependencies before proposing systemic fixes.

Alternative Ecosystem Support

While the current project is heavily optimized for OSS-Fuzz and C/C++ targets, its modular architecture makes it an ideal candidate for cross-language expansion. The system could be extended to support managed languages such as Java, Python, or Rust. This would require the implementation of language-specific code extractors, tailored prompt templates for the LLM agents, and dedicated runtime environments within the local Docker deployment to handle different compilation and fuzzing targets.

Automated Regression Suite Generation

Moving beyond simply fixing the reported crash, the agent framework could be upgraded to generate additional edge-case unit tests. Once a successful patch is validated, an independent LLM tool could analyze the execution trace and generate a regression test suite. This ensures the proposed fix does not inadvertently introduce new vulnerabilities or memory leaks, transitioning the project from a reactive "patching" tool to a "proactive hardening"

framework.

7.2. Methodological Refinements (Should)

Iterative Verification Loops

The system should implement a more robust self-correction mechanism. In the current pipeline, failed patch attempts provide limited context back to the model. By feeding detailed compiler error messages, standard error outputs (stderr), and runtime tracebacks directly back to the LLM agent via LangChain tools, the system can iteratively refine its solutions. This conversational loop mimics a human developer's debugging process and significantly increases the success rate of complex vulnerability resolution.

Hybrid Analysis Integration

To improve precision and reduce computational overhead, the patcher should combine LLM-based code generation with deterministic static analysis tools (such as Clang-Tidy or Facebook Infer). This hybrid approach allows the system to pre-validate generated patches for obvious memory leaks, buffer overflows, or thread-safety violations before pushing them to the computationally expensive containerized execution phase.

Cost-Aware Model Routing

Because the pipeline already supports multiple LLM providers (including OpenAI, Google, and local models via Ollama), it should implement intelligent routing logic. Cost-aware routing would utilize smaller, cheaper, or locally hosted models for "low-confidence" exploration, code parsing, and initial tool usage. Conversely, high-parameter, commercial models would be reserved exclusively for final code generation, complex logic synthesis, and final verification, optimizing the operational cost of the pipeline.

7.3. Infrastructure and Scaling (Next Steps)

Distributed Processing Engine

To handle large-scale vulnerability datasets efficiently, the execution engine needs to transition from a localized Docker build process to a distributed task queue architecture. Utilizing tools such as Celery with Redis, or orchestrating containers via Kubernetes, would enable

the parallel patching of hundreds of OSS-Fuzz issues simultaneously. This is a vital next step for enterprise-grade scalability.

Human-in-the-Loop Integration

While automation is the primary goal, a critical next step is the development of a web-based dashboard for security researchers. This interface would allow developers to manually review, tweak, and approve AI-generated patches. Providing this "trust-but-verify" layer ensures that no hallucinated or suboptimal code is committed directly to production repositories, addressing inherent LLM reliability concerns discussed during client meetings.

Knowledge Base Persistent Memory (RAG)

Implementing a vector database (such as ChromaDB or Pinecone) to store successful previous patches would allow the agent to learn from historical fixes. By using Retrieval-Augmented Generation (RAG), the system could search its persistent memory for similar bug patterns and previous solutions before attempting a fix. This would significantly reduce the "time-to-fix" for recurring vulnerabilities and lower the API token consumption per issue.

8. Conclusion and Evaluation

8.1. Conclusion

This project successfully demonstrated the feasibility of an end-to-end automated pipeline for patching security vulnerabilities in C/C++ projects using Large Language Models. By integrating crash report parsing from OSS-Fuzz, automated reproduction in isolated Docker containers, and multi-model LLM orchestration via LangChain, the team significantly reduced the manual effort required for initial vulnerability triage. The results confirm that AI agents, when provided with high-quality runtime feedback such as compiler errors and sanitizer outputs, can generate viable security patches that adhere to strict project coding standards.

Throughout the development lifecycle, the architecture evolved from a static patch generation pipeline into a dynamic, agent-based system. Initial challenges with pulling volatile pre-built Google container registries necessitated the development of local Docker build integrations. Overcoming these infrastructure hurdles enabled the implementation of an autonomous agent capable of utilizing LangChain tools for folder introspection and iterative code execution. This transition allowed the LLM to actively reason about the codebase and interact with the environment to find bugs, rather than relying solely on zero-shot patch generation.

Ultimately, this work serves as a foundational step toward future self-healing software systems. Minimizing the latency between a vulnerability discovery and its remediation significantly reduces the window of opportunity for attackers. The final pipeline, validated through user testing with PhD researchers, proves that integrating LLMs with traditional fuzzing workflows is a viable strategy for scaling open-source security maintenance.

8.2. Evaluation

The performance and effectiveness of the automated AI patcher were analyzed based on experimental results obtained from the OSS-Fuzz dataset. The system transitioned from testing isolated components to a fully integrated pipeline, allowing for a comprehensive

assessment of both the underlying infrastructure and the AI agent's reasoning capabilities.

Success Metrics

The system was evaluated against three primary criteria:

- **Reproduction Rate:** The ability of the Docker-based runtime to successfully trigger the reported crash locally using the provided testcase, bypassing the reliance on external container registries.
- **Compilation Rate:** The percentage of AI-generated patches that successfully pass the target project's build system without syntax errors.
- **Patch Quality:** The ratio of patches that not only compile but also resolve the crash (verified by the `patch_quality.py` pipeline) without regressing existing functionality.

Comparative Analysis

The modular architecture facilitated a direct comparison between different Large Language Models. Preliminary results indicate that local models executed via Ollama provide zero-cost execution and high privacy, which is highly advantageous for proprietary codebases.
However, frontier models via OpenAI and Google demonstrate superior reasoning capabilities in complex memory corruption scenarios, such as out-of-bounds reads or use-after-free vulnerabilities. The integration of LangChain tools further amplified these reasoning capabilities, allowing advanced models to effectively navigate the project structure and apply contextual fixes.

8.3. Team Collaboration and Methodology Evaluation

The project was managed using the Agile Design methodology, characterized by two-week sprints and weekly client feedback sessions. While the framework provided a solid structure, the team encountered challenges regarding member availability and synchronized communication, primarily due to overlapping commitments with other academic modules.

To mitigate these risks, the team leveraged the flexibility of the Scrum framework:

- **Task Distribution:** Requirements were divided into granular tasks and distributed evenly among the six members to ensure parallel progress.
- **Role Rotation:** The Scrum Master role was rotated weekly, allowing every member to coordinate stand-up meetings and identify individual bottlenecks early.
- **Collaborative Problem Solving:** When technical issues appeared the team tried to shift from individual task focus to collaborative research.
- **Communication Channels:** Direct and frequent communication through dedicated channels allowed for quick resolution of conflicts and ensured that all deliverables met the quality standards wanted by the team and clients.

Ultimately, the combination of iterative feedback and adaptive task management enabled the team to achieve all the necessary requirements and some additional ones as well, making the team collaboration a success.

Appendix

A. Additional tables and figures

This appendix contains the full-resolution UML diagrams referenced in Section 3.2.3.

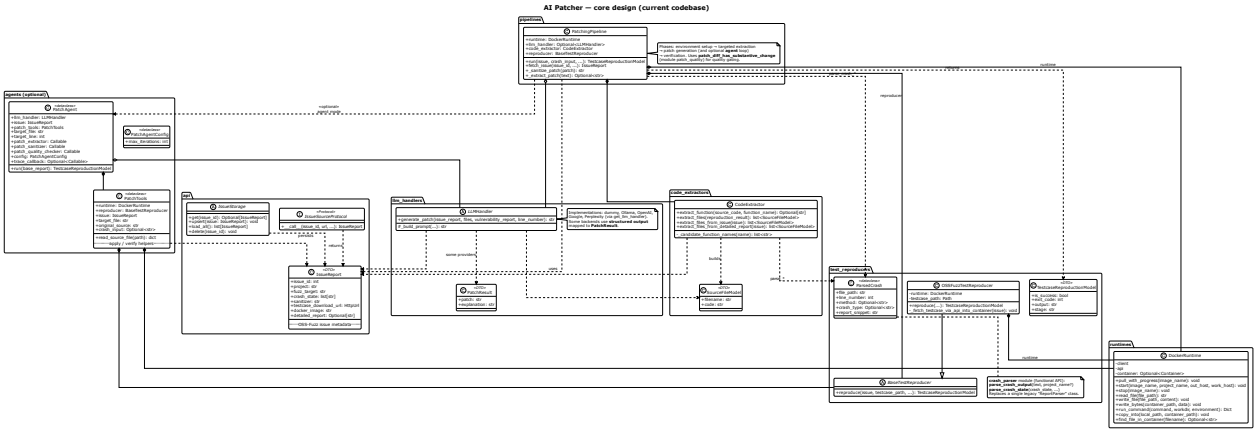


Figure 1: System architecture class diagram

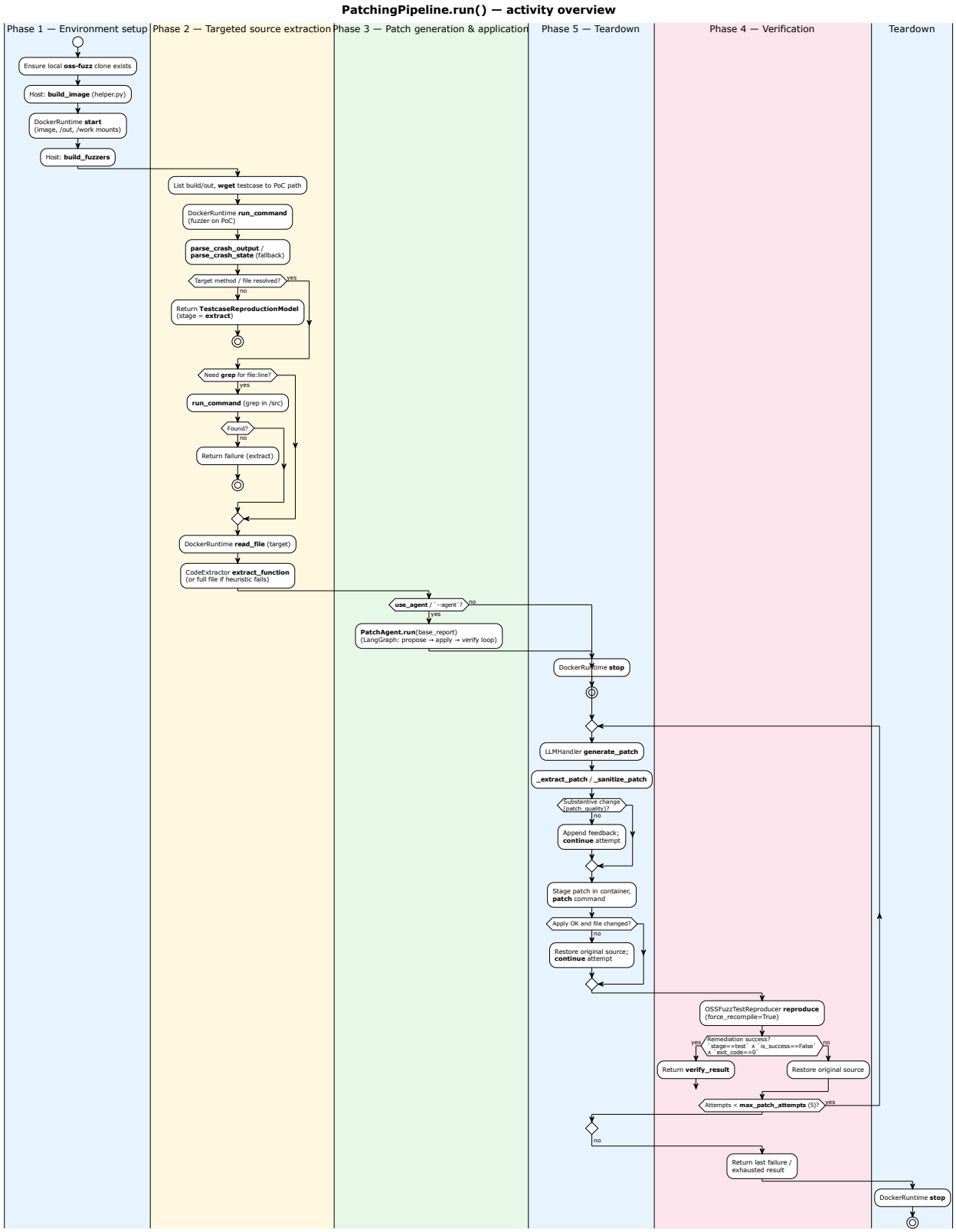


Figure 2: AI Patcher activity diagram

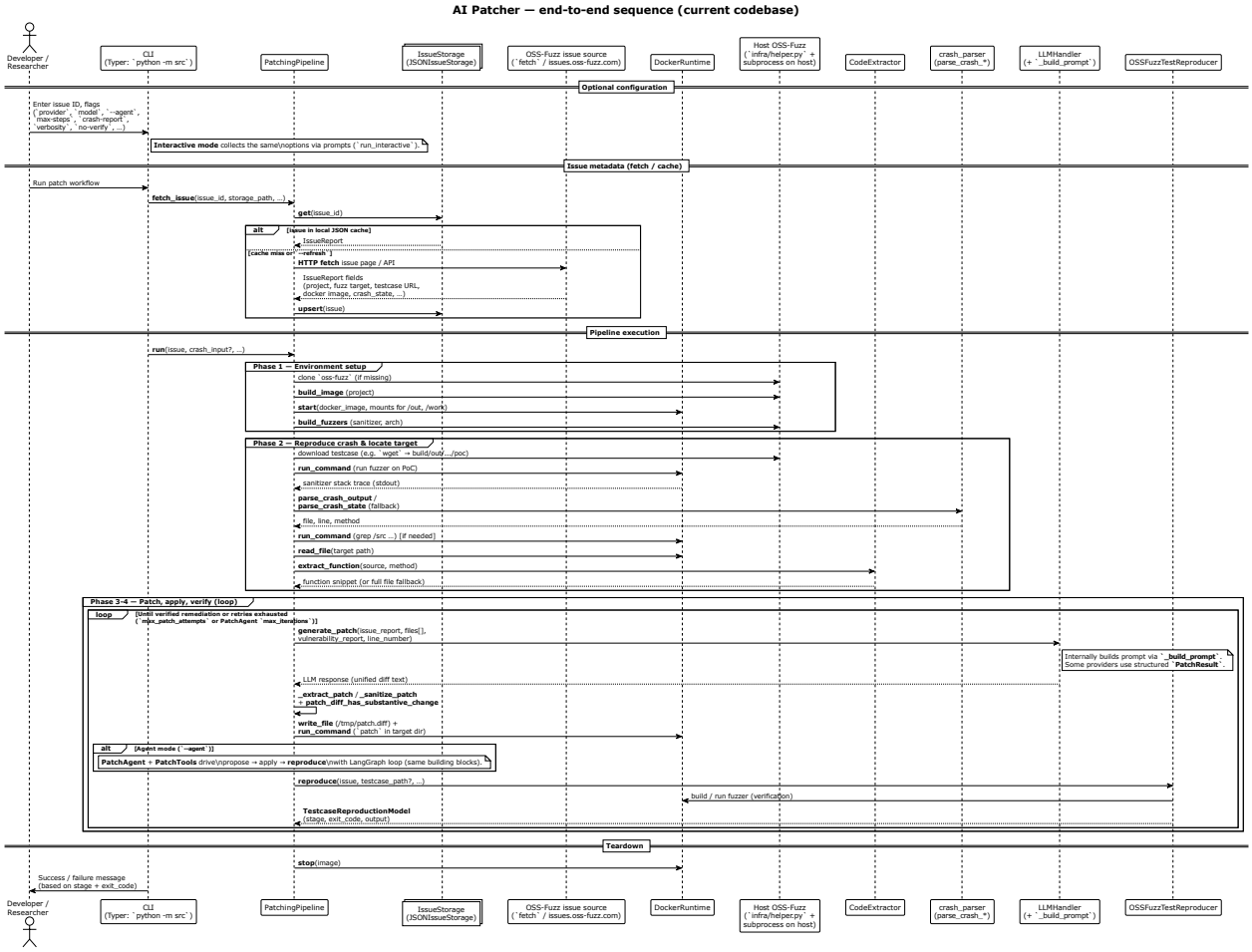


Figure 3: AI Patcher sequence diagram

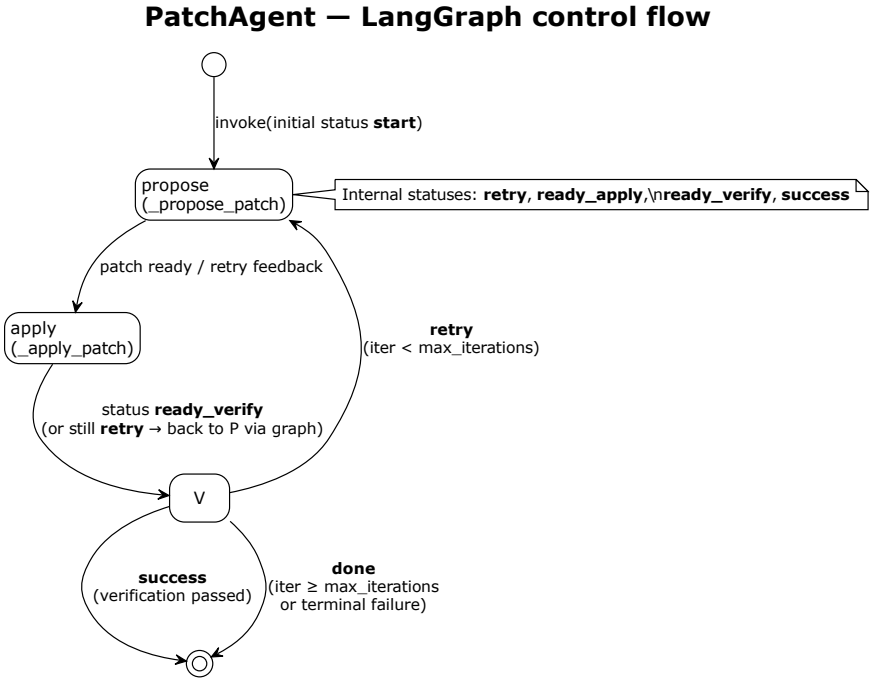


Figure 4: AI Patcher state machine PatchAgent diagram

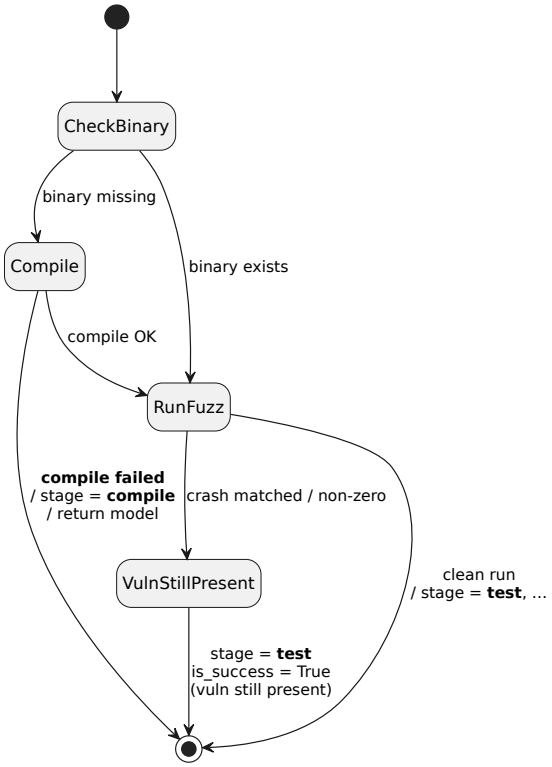


Figure 5: AI Patcher state machine verification diagram

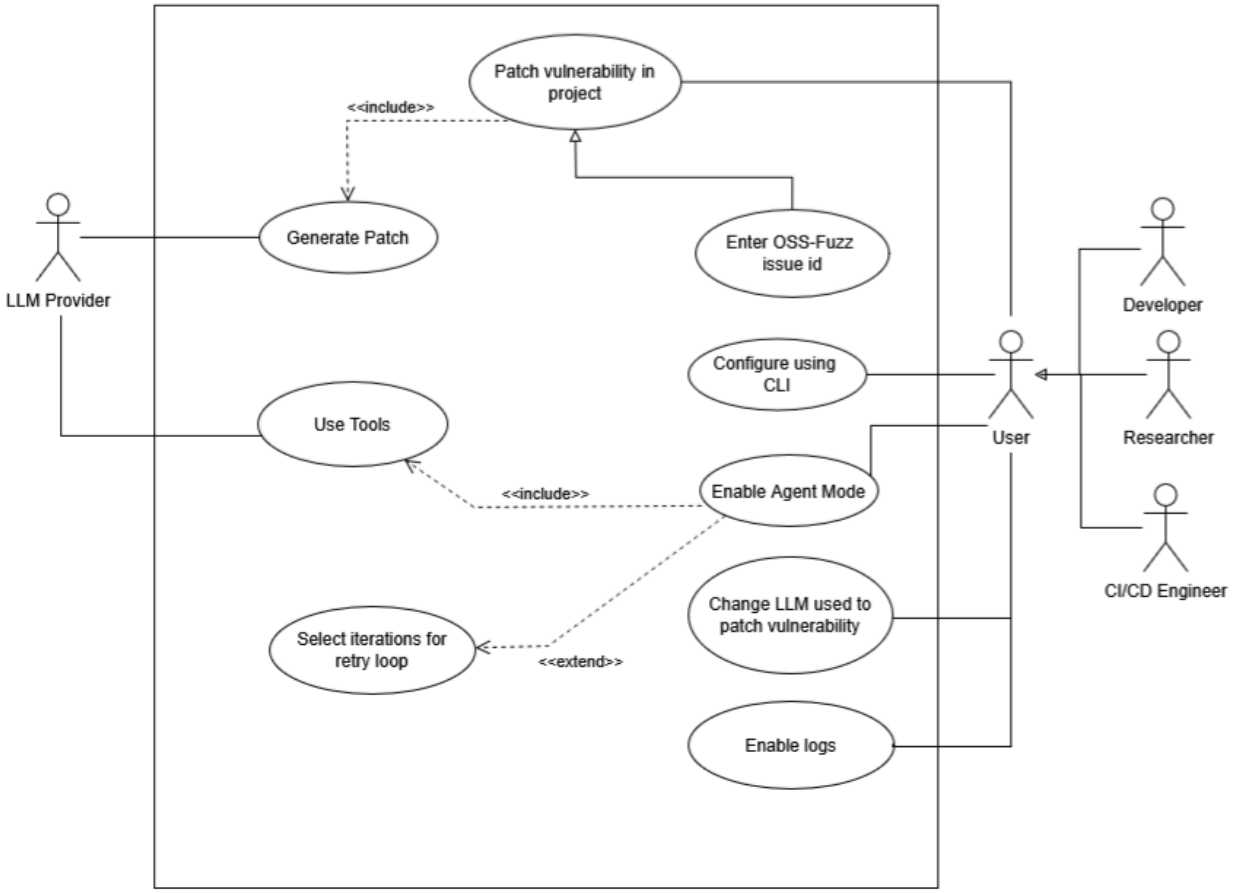


Figure 6: AI Patcher use case diagram

B. Contribution

The following table shows the contribution of each member:

Task	**Contributor**
The system shall have no memory leaks which can be exploited to run arbitrary code	Ion and Andrey
The system shall be optimized to minimize execution time.	Andrey, Benjamin and Roman
The system shall be designed to easily add more analysis and tools.	Ion, Andrey, Benjamin and Roman
The system shall minimize the number of queries made to the LLM to reduce costs and latency.	Roman
The system shall be able to run on any Operating System	Ion, Andrey, Benjamin and Roman
The system shall have documentation on how to run the pipeline.	Benjamin
The system shall have documentation on how to configure the pipeline.	Benjamin
The system shall have a well-documented codebase to facilitate future extensions and maintenance	Ion and Benjamin
Set up GitLab	Ion
Conduct initial research and hands-on exercises for OSS-Fuzz report parsing, LangChain, and Docker interaction to mitigate inexperience risk.	All members
The system architecture shall be designed in modular components with clear interfaces to ensure maintainability.	Ion, Andrey and Benjamin

Task	**Contributor**
The system should output the specific lines changed (show the differences)	Benjamin
The system shall be able to read and process data and user inputs in under a minute.	Andrey and Ion
The system must get the sanitizer report locally	Ion
The system should provide tools to AI for making the pipeline autonomous	Benjamin
The system should implement a feedback loop	Benjamin
Finalize the high-level system architecture and design of modular components.	Furqan, Thijmen and Benjamin
The system must fetch project name from the given OSS-Fuzz issue ID	Ion and Benjamin
The system must fetch fuzz target from the given OSS-Fuzz issue ID	Roman
The system must fetch the reproducer test case from the given OSS-Fuzz issue ID	Ion and Benjamin
The system must be able to recognize broken or invalid inputs and notify the user immediately.	Ion and Andrey
The system must use the fetched information to start a docker container	Benjamin, Ion and Andrey
The system must accept OSS-Fuzz issue ID input from the user.	Roman
The system shall abstract all Large Language Model (LLM) interactions behind a generic interface.	Benjamin
Check Peerpulse reports	Andrey
LLM - Custom prompts	Thijmen

Task	**Contributor**
Docker - read files	Andrey
Docker - compile and get failing files	Benjamin
LLM - Combining with files read	Furqan
Pipeline integration	Andrey
The system must identify the buggy file and the buggy function to be modified	Ion
The system must generate a prompt containing all necessary details to query an LLM for a code fix.	Thijmen
The system must be operable using a Command Line Interface (CLI)	Andrey
Write the Reflection report	Furqan and Thijmen
The system must be able to recompile the project	Ion
The system must be able to modify the source code of the OSS-Fuzz project by applying the patch received from the LLM.	Andrey
The system must be able to run fuzzing tests to verify the fix.	Benjamin
The system must return a status action (Success / Failed)	Roman
The system should allow the user to change the LLM	Benjamin
The system should have a configuration argument in the CLI	Ion
The system shall handle build errors within the pipeline gracefully and provide clear, actionable error messages to the user.	Andrey
Poster Design	Furqan and Thijmen
Final Design Report	Everyone
Final presentation preparation	Everyone

Table 7: Tasks mapped to their contributor

C. Meetings with Clients

B.1 Meeting 06.02.2026

Goal:

Our first meeting with the supervisors and clients. We wanted to get a general overview of the project and formulate the requirements.

Meeting time: 1 hour

Notes:

The group has asked various questions and found out what the project is about from the authors themselves. We were able to formulate the task, must-have requirements and future improvements that may help the client. The goal of the project is to build a program that can automatically fix vulnerabilities using LLMs. Google has built OSS Fuzz, an open-source platform where people can find bugs and vulnerabilities using fuzzing, a technique when various randomized inputs are fed into the program with hope that one of them crashes or stalls the application. Those vulnerabilities are typically fixed manually, but the client wants an application which can use an LLM and other algorithms to try to fix the errors automatically. If successful, that can decrease the overall pool of vulnerabilities along with bugs found in open-source programs. The team has found that the program must be extensible and modular, for a possibility to change some parts of the application or try different ways of solving the same problem. With that in mind, the group has started to draw some basic architecture and try to understand the problem deeper. At the end of the meeting, we had a full usage diagram, the requirements and a plan for the next week.

Problems:

The main problem was the access to OSS Fuzz, which we did not have. Luckily, the group has established a direct communication channel with a client, and we were able to quickly find another platform that does not have closed access and can show us all the issues straight away.

Summary:

The group has met with the clients and formulated the requirements for the project. We

discussed the architecture, overall plan, and a way to tackle the project from the start. We have also received access to OSS Fuzz and the documentation. The team has assigned tasks for the following week and established ways to communicate to clients and supervisors.

B.2 Meeting 13.02.2026

Goal:

Our second meeting with the supervisors where we wanted to present our draft project report, as well as ask other questions related to the actual technology.

Meeting time: 1 hour

Notes:

Before the meeting the team submitted a project proposition to the clients, our report contained MOSCOW requirements, functional and non-functional requirements, as well as other parts such as some user-stories. During the meeting the team wanted to receive some feedback to better structure and rewrite the proposition. The clients have shown us an example report and started talking about concrete fixes that we could implement. Some of those fixes were related to the project structure, while some of the other ones have been linked to the technology underneath OSS Fuzz. Overall, we received a lot of feedback along with an example report from the previous year. Some of the comments included a lack of testing description, risks description and a more specific task distribution within the project. Most of the time has been spent on the technical side of the proposal, and the client has shown the team an example fuzzed project and how to run it.

Problems:

The two problems that have slowed down the team were both related to the report. Firstly, the team was not able to realize what was needed to be in the project proposition. There was no template, and the first meeting was mostly related to talking about the technical requirements, rather than focusing on the structure of the report. This is why the team has misunderstood what was needed of us and partially failed to deliver a perfect report. The second problem was also related to the report, but some of the technical comments could have been eliminated by having better communication within the team. Since we have decided to split the group into people who write the project proposition and a part of the team

responsible for researching the documentation. After that, we did not consider rewriting the proposition which resulted in some misunderstanding.

Summary:

The team has presented the first project proposition to the clients and received some good feedback from them. The feedback consisted of both structural and technical comments. We have discussed some possible problems with our requirements and concluded that a new version of the report is going to be presented by the next meeting.

B.3 Meeting 20.02.2026

Goal:

Our third meeting with the clients, where we showed our first prototype and finished the project proposal.

Meeting time: 1 hour

Notes:

The meeting has started with a group talking about the diagrams and current progress. Part of the group was online due to the upcoming holidays, but that did not stop them from clearly explaining the diagrams along with helping the discussion. Main topics of discussion were related to the use-case diagram that was presented by the group. Later, the team showed clients their current progress with the prototype. It could read the report from OSS Fuzz and easily present the information to the clients. We have discussed future improvements such as error handling, retries and ways to display and input information.

Problems:

There were no significant problems aside from some comments on the diagrams. Those comments related to the actual technology stack, and the team has fixed them shortly after.

Summary:

The team has presented a first prototype of the technology and has finished the proposal.

B.4 Meeting 06.03.2026

Goal: Show current progress and ask how to start our pipeline.

Meeting time: 1 hour

Notes: The group has completed separate components, but failed to connect them together. We have a working Docker environment, AI modules, but lack understanding in actually connecting it all together. OSS fuzz provided some sample code, but we could not use it directly and decided to move some modules for starting and testing the program inside of our repository. However, the team was not able to actually reproduce the error, and all separate components were rendered useless.

Problems: The team did not spend enough time on the actual error reproduction of the error, but rather separated into different components. We were not able to show how different components interact together, resulting in an unproductive meeting. However, we were able to ask a lot about the actual problems and solve a lot of hurdles that were unclear in the project and documentation of OSS Fuzz.

Summary: The team has presented separate components and showed overall progress with the meeting taking place in more of a question-answer based approach. We have talked a lot about the actual methods and ways of error reproduction in OSS Fuzz, as well as decided on how to build local containers.

B.5 Meeting 13.03.2026

Goal: Show current progress and find out more about the tools used in LangChain.

Meeting time: 1 hour

Notes: The team wanted to show the progress in the pipeline and ask more questions regarding the tools section of LangChain and how it can be used. Currently, the components were refactored to work better and we could start some docker containers, however, the main issue lied in the connection of separate components and building of Docker containers. The team tried to pull containers from Google container registry, however, as we have found out they delete artifacts at different times, so most projects cannot rely on that. The team decided to try and build containers locally, but was not able to finish the full build integration. That is why we talked to the client and tried to ask more questions regarding the building of the containers, reproduction of an error and other stuff regarding reliability of LLMs.
The team has also asked about the tools section, and decided to implement an Agent mode, which could use different LangChain tools to automatically run useful commands without a

strict pipeline. However, for the next sprint the group will focus on the normal reproduction pipeline and try to build the Docker containers locally.

Problems: The main problem was that some Docker containers were not available to pull, and the team did not have enough time to implement the building of the containers locally, and that is why the pipeline was still not able to run properly. However, we talked a lot about the reproduction and different ways of building the containers and have created a plan to tackle that problem.

Summary: The team has shown the current progress with improved LLM modules as well as talked about the local Docker build that we wanted to focus on next week. We have also discussed LangChain tools and decided to research that in more detail.

B.6 Meeting 20.03.2026

Goal: Introduce a fully working pipeline and ask questions regarding the LLM stability

Meeting time: 1 hour

Notes: The team has a fully finished pipeline now, and we showed it to the client. The team has presented a fully working CLI with better options, as well as a fully working pipeline that could build containers locally. The program still has some bugs, but most of them are either related to lack of hardened testing that the team wants to focus on later, or on the LLMs not being able to fix the vulnerabilities present in OSS fuzz containers. After a talk with the client, the team has concluded that the latter issue is related more to the prompts, temperature and randomized outputs that LLMs typically give out. Overall, we have presented an almost fully working product with some bugs still present.
However, the client wanted us to focus more on the tools within LangChain that could later be used by an LLM to run arbitrary tools that it may need to find bugs. The main idea of tools is to allow the LLM to do more than just generating patches, but rather reason and use the tools that it may need for context and patching overall.

Problems: There were no serious problems within the pipeline, however, the team was not able to fix all of the bugs and implement Langchain tools, so we would focus on that in the next sprints.

Summary: The team has shown a fully working pipeline as well as talked about different

ways of using LangChain tools in order to fix the OSS fuzz error automatically. We have shown a better CLI and improved the overall pipeline.

B.7 Meeting 01.04.2026

Goal: Show a new agent pipeline and talk about user-testing.

Meeting time: 1 hour

Notes: The team has created a tools pipeline which allows the application to use available tools such as folder introspection and docker containers directly from the LLM. We have shown how it works and the client was able to ask many questions regarding the processes and ways our tools are employed by the pipeline. We have also discussed user testing, and confirmed that it will be conducted next week with some of the PhD workers. The team has also scheduled a final presentation.

Problems: There were no problems for this meeting.

Summary: The team has presented a fully working tool-based variant of the application and talked about user-testing.

B.8 Meeting 10.04.2026

User testing: We performed some user test with volunteers provided by the client. The user testing notes are restructured and added above in the testing section.

D. AI usage in project

We utilized ChatGPT to help us understand the project better and to grasp how to set up OSS-Fuzz and configure Docker environments. Additionally, it assisted us in debugging the code, while at the same time enhancing the clarity and structure of our documentation (including this report). All outputs generated by the AI were thoroughly reviewed and modified by the team before being incorporated into the project. The team remains fully responsible for the accuracy, originality, and overall content of this project.