Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Integrating LLM based Automated Bug Fixing into Continuous Integration - Analysis of Potentials and Limitations

Abschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft (HTW) Berlin
Fachbereich 4: Informatik, Kommunikation und Wirtschaft
Studiengang *Internationale Medieninformatik*

1. Gutachter_in: Prof. Dr. Gefei Zhang
2. Gutachter_in: Stephan Lindauer

Eingereicht von Justin Gebert [s0583511]

22.07.2025

Acknowledgements

Abstract

Generative artificial intelligence is reshaping software engineering practices by enhancing and automating core tasks, including code generation, debugging and program repair. Despite these advancements, existing automated program repair (APR) approaches suffer from complexity, high computational demands, and a lack of integration within practical software development lifecycles.

In this thesis, we address these challenges by introducing a novel and lightweight automated bug fixing system leveraging Large Language Models (LLMs), explicitly designed for seamless integration into continuous integration pipelines. Our containerized solution automates the bug-fixing lifecycle, from GitHub issue creation to the generation and validation of pull requests, reducing manual intervention and streamlining development workflows.

We evaluated our approach using the QuixBugs benchmark, testing twelve LLMs for effectiveness, efficiency, and costs. The results demonstrate repair success rates of up to 100% with short execution times and low costs, highlighting the practicality of our streamlined solution in effectively repairing small-scale software bugs.

The outcomes underscore the feasibility and potential of integrating APR directly into continuous integration pipelines. They also show current limitations of LLM-based automatic bug fixing. Enhancements like adaptive model selection and improved integration techniques are outlined to further refine and optimize this promising approach in the future.

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

GenAI generative artificial intelligence 1
AI artificial intelligence 7
NLP natural language processing 7
APR automated program repair 1
LLM large Language model 1
CI continuous integration 1
SDLC software development lifecycle 1
RAG Retrieval Augmented Generation 8
UI user interface 35

1. Introduction

Generative artificial intelligence (GenAI) is rapidly changing the software industry and how software is developed and maintained. The emergence of large Language models (LLMs), a subfield of GenAI, has opened up new opportunities for enhancing and automating various domains of the software development lifecycle (SDLC). Due to their remarkable capabilities in understanding and generating code, LLMs have become valuable tools for developers. Everyday tasks such as code generation, refactoring and debugging can be enhanced by using LLMs [1, 2].

Despite these advances, fixing bugs remains a challenging and resource-intensive task, often negatively perceived by developers [3]. It can cause frequent interruptions and context switching, resulting in reduced developer productivity [4]. Fixing these bugs can be time-consuming, leading to delays in software delivery and increased costs. Software bugs harm software quality by causing crashes, vulnerabilities, or even data loss [5]. In fact, according to the Consortium for IT Software Quality (CISQ), poor software quality cost the U.S. economy over $2.4 trillion in 2022, with $607 billion spent on finding and repairing bugs [6].

Given the critical role of debugging and bug fixing in software development, automated program repair (APR) has gained significant interest in research and industry. The goal of APR is to automate the complex process of bug fixing [1], which typically involves localization, repair, and validation [7, 8, 9, 10, 11]. Recent research has shown that LLMs can effectively enhance automated bug fixing, thereby introducing new standards in the APR world and showing potential for significant improvements in the efficiency of the software development process [9, 12, 13, 14, 15, 16].

However, existing APR approaches are often complex and require significant computational resources [17], making them less applicable in budget-constrained environments or for individual developers. Additionally, the lack of integration with existing software development tooling and lifecycles limits their practical applicability in real-world development environments [18, 12].

Motivated by these challenges, this thesis explores the potential of integrating LLM-based automated bug fixing into existing software development workflows using continuous integration (CI) pipelines. CI is the backbone of modern software development, ensuring rapid and reliable software releases [19]. By leveraging the capabilities of LLMs, we aim to develop a lightweight and cost-effective prototype for automated bug fixing that seamlessly integrates with CI pipelines. Considering computational demands, the complexity of integration, and practical constraints, we aim to provide insights into the possibilities and limitations of our approach by answering the following research questions:

- **RQ1:** How can LLM-based automated bug fixing be effectively and efficiently integrated into a CI pipeline?

- **RQ2:** What are the potentials and limitations of this integrated approach with respect to repair success rate, execution times and cost-effectiveness?

The thesis is organized as follows:

Chapter 2 provides theoretical background on software development, GenAI in the context of software development, and APR. Chapter 3 outlines the methodology used for this thesis, consisting of preparation, implementation, and evaluation. Chapter 4 showcases the functional and non-functional requirements constructed for the prototype. Chapter 5 explains the implementation and resulting system in detail. Chapter 6 showcases the resulting workflow when using the prototype and presents the results of the evaluation. Chapter 7 discusses the potentials and limitations of the prototype based on the results. Finally, chapter 8 concludes the thesis by summarizing the findings and contributions of this work.

2. Background and Related Work

In this chapter, the required theoretical background and context for this thesis are explained. First, fundamental concepts of software development, the Agile SDLC, continuous integration (CI), and software project hosting platforms are introduced. The second part explores GenAI and LLMs with their rising role in software development practices. The third part examines the evolution and current state of APR with examples of existing approaches.

2.1. Software Development

This section introduces core concepts of software development, starting with the SDLC, followed by the importance of CI in modern software development, and the role of software project hosting platforms.

2.1.1. Software Development Lifecycle

Engineering and developing software is a complex process, consisting of multiple different tasks. To structure this process, SDLC models have been introduced. These frameworks constantly evolve to adapt to the changing needs of software development. One of the most promising and widely used SDLC models today is Agile [20, 21].

The Agile lifecycle introduces an iterative approach to software development, focusing on collaboration, feedback, and adaptability. The goal of Agile is frequent delivery of small functional software features, allowing continuous improvement and adaptation to changing requirements [20, 21]. Frameworks like Scrum or Kanban are used to apply Agile in a development environment [22].

An Agile iteration consists of multiple stages, each contributing to the overall development cycle. Figure 2.1 visualizes an example of an Agile iteration interpretation. Iterations start with a planning phase where requirements are gathered and prioritized. Secondly, the architecture and design of the required changes are constructed in the design phase. The third stage involves developing the prioritized requirements. After development, the changes are tested for issues or bugs in the testing stage. Upon successful integration and testing, the changes are released in the deployment stage. Finally, internal and user feedback is collected for review [23].

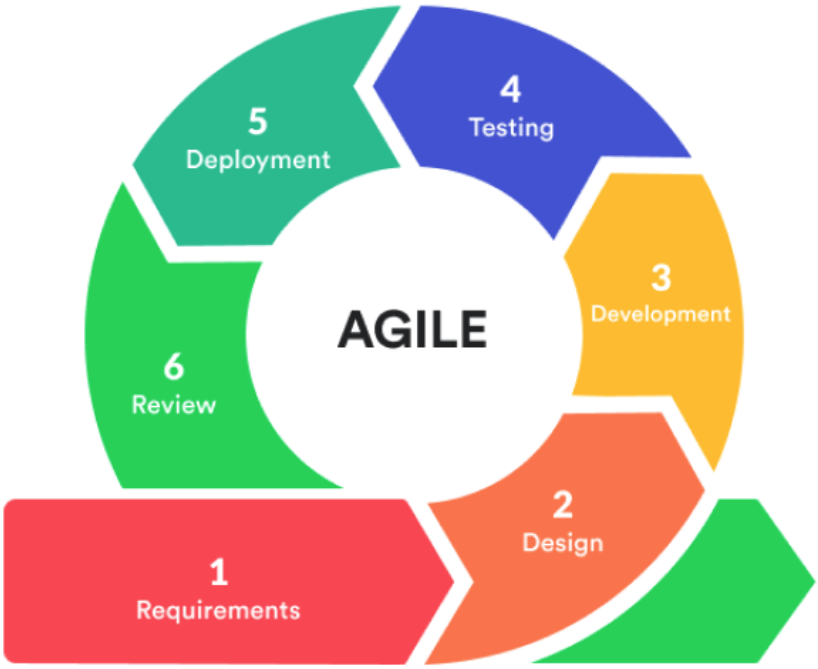

Figure 2.1.: *Agile software development lifecycle, Source: [24]*

When bugs arise during an iteration, requirements can be reprioritized, and the iteration can be adapted to fix these issues. This adaptability is a key feature of Agile software development, allowing teams to quickly respond to changing requirements and issues. However, this can slow down the delivery of planned features.

Modern software systems are moving towards loosely coupled microservice architectures, resulting in more repositories of smaller scale, tailored towards specialized domains. This trend is driven by the need for flexibility, scalability, and faster development cycles. This approach aligns with modern Agile software development practices [25]. Along with this trend, developers tend to work on multiple projects simultaneously, which can lead to more interruptions and context switching when problems arise and priorities shift [26, 4].

2.1.2. Continuous Integration

Continuous integration (CI) has become a standard practice in Agile software development to accelerate development and delivery. CI enables frequent code integration into a repository, by automating steps like building and testing, thus providing rapid feedback right where the changes are committed. This supports critical aspects of Agile software development, enhancing delivery, feedback, and collaboration [19]. Figure 2.2 illustrates a typical CI cycle, where code changes are automatically fetched from source control, built, and tested and the results are reported back to developers.

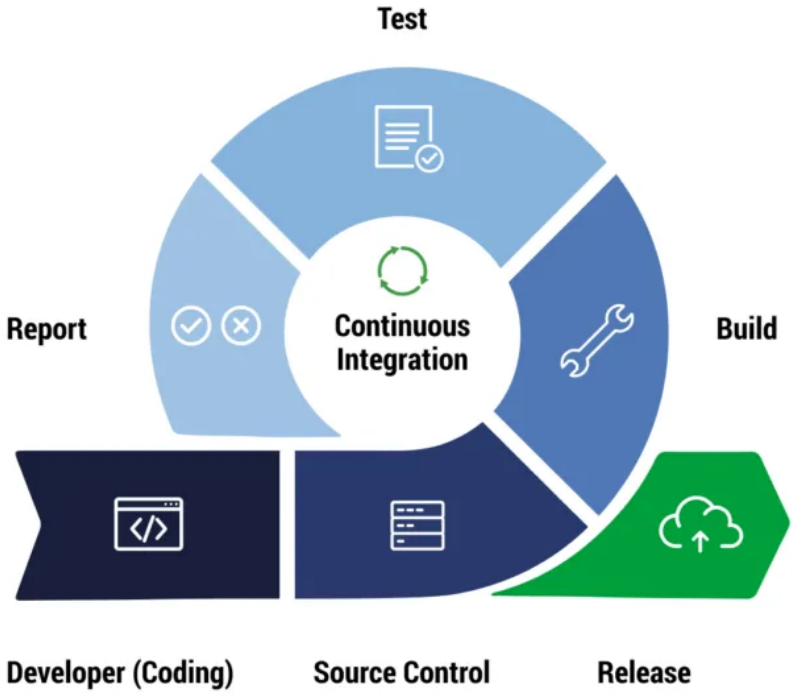

Figure 2.2.: *Continuous Integration cycle, Source: [27]*

Although CI improves feedback loops, it can also result in additional overhead. Effort and infrastructure must be invested to keep pipelines running [28], and projects may suffer from long build times that harm developer productivity [29].

2.1.3. Software Project Hosting Platforms

Modern software projects are typically hosted on platforms such as GitHub or GitLab. GitHub is the largest, with over 100 million developers and more than 518 million repositories, making it the world's leading open-source community [30].

These development platforms offer tools and services for the entire SDLC, including project hosting, version control, issue tracking, bug reporting, project management, backups, collaborative workflows, and documentation capabilities [31, 21].

A key feature of GitHub is GitHub issues, this feature enables backlogs and tracking of tasks, features and bugs for individual repositories. Issues can be created, assigned, labeled, and commented on by everyone working on a codebase. This feature provides a structured way to manage and prioritize work within a project [32]. Figure 2.3 shows an example of a GitHub issue.

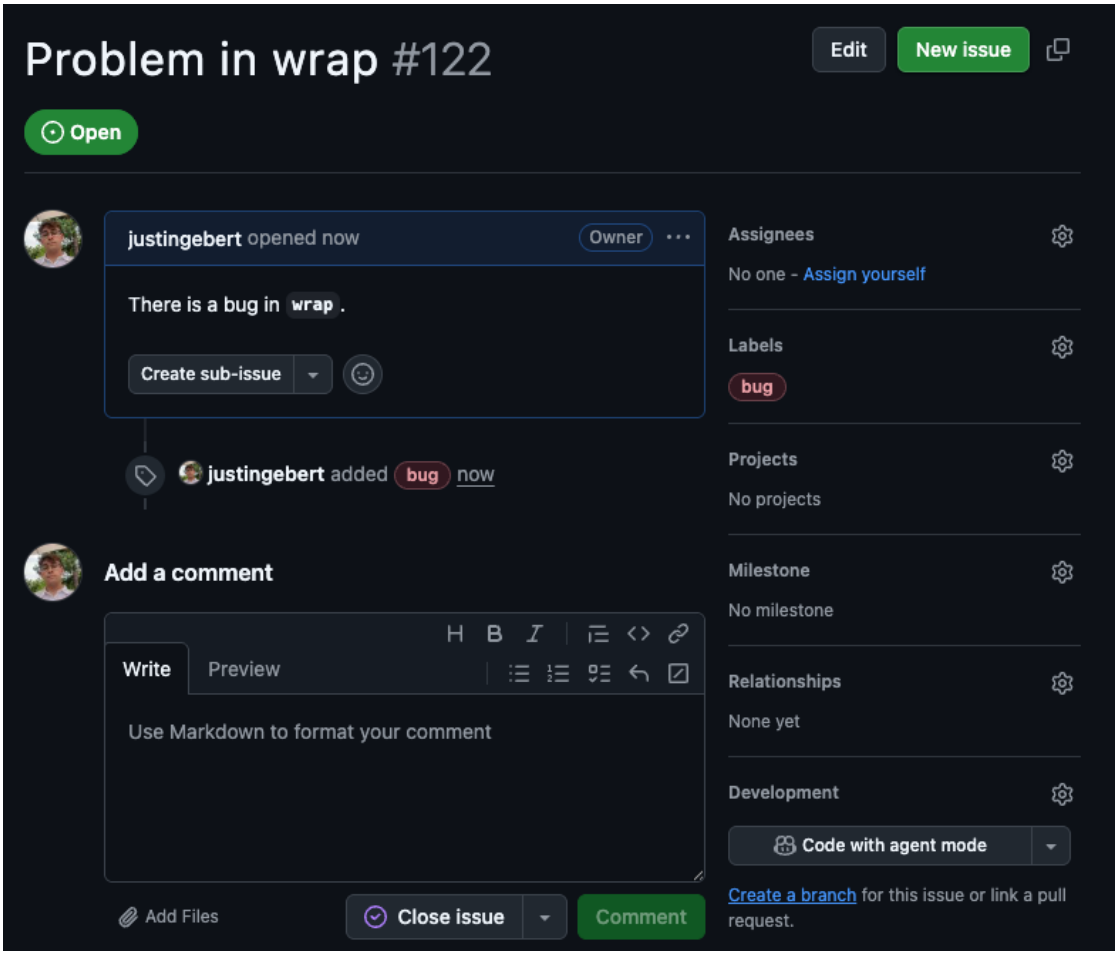

Figure 2.3.: *Example of a GitHub, Source: screenshot*

For integrating and reviewing code, GitHub provides pull requests. A pull request proposes changes to the codebase, integrating a review process to validate changes before merging into the production codebase. Code changes are displayed in a diff format[1], allowing reviewers to examine the changes made. This process is essential for maintaining code quality and ensuring that changes are validated before merging. Pull requests can be linked to Issues, allowing easy tracking of changes related to specific tasks or bugs [34].

Additionally, GitHub offers a managed solution for running CI pipelines called GitHub Actions. These pipelines are configured using workflow files written in YAML[2]. Workflows can run on either self-hosted or GitHub-hosted runners. Each workflow is defined by triggers, jobs, and steps. Specific events trigger the workflow, which then executes one or more jobs, each composed of multiple steps [36]. Figure 2.4 illustrates the components of a GitHub workflow.

[1] A diff format shows the difference between two files by highlighting added, removed, or modified lines [33].
[2] YAML is a human-readable data serialization language [35].

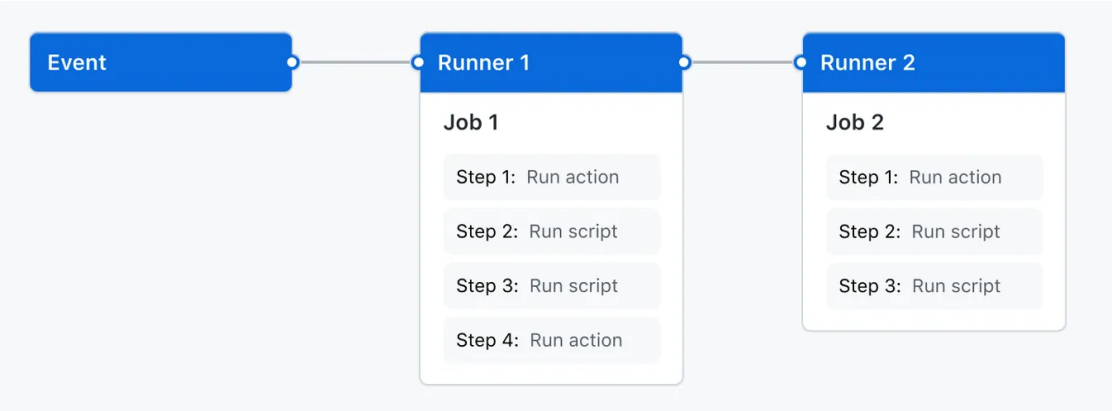

Figure 2.4.: *Components of a GitHub Action, Source: [36]*

The results and logs of a workflow can be viewed from multiple places in the GitHub User Interface, including the Actions tab, Pull Request page, and the repository's main page. This integration provides a seamless experience for developers to monitor and manage their CI processes directly within a repository [37].

2.2. Generative AI in Software Development

This section examines the role of GenAI in modern software development. First, GenAI and LLMs are defined. The second part focuses on the impact of GenAI on software development practices.

2.2.1. Generative AI and Large Language Models

GenAI is a subfield of artificial intelligence (AI) referring to systems that generate new content based on patterns learned from extensive training data. Recent advances in machine learning, especially deep learning, have made it possible for generative systems to produce text, images, or code that resembles human-created content [38].

The transformer architecture marked a major breakthrough in text generation and natural language processing (NLP) research. This architecture lays the groundwork for Large Language Models [39, 40]. Extensive training, results in LLMs with billions of parameters, allowing them to understand and generate text in diverse natural and programming languages. Research has shown that a model's size directly impacts its performance, with larger models generally achieving better results in various NLP tasks [41]. However, training and operating larger models requires significant computational resources [42, 40]. Furthermore, despite modern LLMs showing promising results in text generation, they can still hallucinate incorrect or biased content [42].

When using a LLM, an input prompt must be provided. Designing this input is a process known as prompt engineering. The quality and specificity of the prompt directly influence the model's output. In zero-shot prompting, the model is given only the task description, while in few-shot prompting, a few relevant examples are included

to better guide the model's behavior. Brown et al. [43] showed that few-shot prompting can remarkably improve model performance across diverse tasks.

Text used for input and output is tokenized, meaning the text is broken down into smaller units (tokens) for processing. The input is constrained by a model's context window, which is the maximum amount of text the model can process at once [40].

LLMs can be accessed via APIs offered by providers like OpenAI, Anthropic, or Google. A selection of LLMs with their characteristics is listed in Section 3.1.2.

2.2.2. Large Language Models in Software Development

LLMs are reshaping software development by automating various tasks [1]. With billions of parameters and pre-training on massive codebases, these models show extraordinary capabilities in this area [18]. Tools like ChatGPT [44] and GitHub Copilot [45] have become popular in the software development community, providing developers with AI-powered code suggestions and completions [46]. These tools are applied in various stages of the SDLC, including requirements engineering, code generation, refactoring, testing, and debugging [1, 2, 46]. By using LLMs for these tasks, development cycle times can be reduced by up to 30 percent [46, 47]. Furthermore, research has shown that these tools positively impact developer satisfaction and reduce cognitive load [47].

Despite the rapid adoption of GenAI in many areas of software development, this technology still faces limitations. LLMs face difficulties with tasks that are outside of their training scope or require specific domain knowledge [1]. Limited context windows create challenges when working with large codebases and complex projects, restricting true contextual or business requirement understanding [46]. When generating code, LLMs can produce incorrect or insecure outputs, leading to unexpected bugs or vulnerabilities [1, 46]. Additionally, integrating LLMs can introduce vulnerabilities threats like prompt injection, where malicious instructions are injected and can lead to generation of harmful code [48]. Moreover, code generated by LLMs is based on existing training data, raising questions about ownership, responsibility, and intellectual property rights [49, 1].

Facing these challenges, different approaches are actively being developed and researched, including AI Agents [12, 13], Retrieval Augmented Generation (RAG) approaches [9], and interactive systems [15]. These paradigms aim to enhance LLM capabilities by providing additional context, enabling multi-step reasoning, or allowing interactive feedback loops during code generation and debugging [1, 2]. Section 2.3.1 discusses the mentioned approaches in more detail in the context of APR.

Recent research is exploring solutions which integrate LLMs into existing software development practices and workflows, leveraging existing development tools and platforms for seamless integration into the SDLC [2, 45, 50, 49].

2.3. Automated Program Repair

APR describes software used to detect and repair bugs in codebases with minimal human intervention [51]. The goal of APR is to automate the bug-fixing process, reducing workload for developers and allowing them to focus on more relevant tasks [1].

APR systems fix specific bugs by applying patches, typically generated using a three-stage approach: localizing the bug, repairing it, and finally validating the fix [51, 52]. This approach mirrors a developer's bug-fixing process, where the bug is identified, fixed, and then tested and reviewed to ensure the fix works as intended [13]. GetaFix is a prominent example of an APR system applied at scale. It is used at Meta to automatically fix common bugs in their production codebases. [52]

The field of APR has greatly benefited from rapid advancements in AI, with new research and benchmarks continually setting higher standards [2, 1].

This section provides an overview of the evolution of APR, related work, and the current state of APR systems, followed by a selected list of APR benchmarks used in research and industry.

2.3.1. Evolution of Automated Program Repair Techniques

APR has experienced multiple paradigm shifts over the years, categorized into key stages marked by significant advancements in techniques and methodologies.

Traditional Approaches:

Traditional APR approaches typically rely on manually crafted rules and predefined patterns [12, 53, 54]. These methods can be classified into three main categories: search-based, constraint/semantic-based, and template-based repair techniques.

- **Search-based repair** searches for the correct predefined patch within a large search space [12, 16, 10]. A popular example is GenProg, which uses genetic algorithms to evolve patches by mutating existing code and selecting patches based on fitness determined by test cases [55].
- **Constraint/Semantic-based repair** synthesizes patches using constraint solvers derived from the program's semantic information and test cases [12, 56]. Angelix is a prominent example of this approach [56].
- **Template-based repair** relies on mined templates for transformations of known bugs [53]. These templates are mined from previous human-developed bug fixes [53, 54]. GetaFix is an industrially deployed tool, learning recurring fix patterns from past fixes [52].

These traditional approaches face significant limitations in scalability and adaptability. They struggle to generalize to new and unseen bugs or adapt to evolving codebases, often requiring extensive computational resources and manual effort [2, 15].

Learning-based Approaches:

Machine learning techniques introduced learning-based APR approaches, increasing the variety and number of bugs that can be fixed. Deep neural networks leverage bug-fixing patterns from historical fixes as training data to learn how to generate patches and translate buggy code into correct code [53, 57]. Prominent examples include CoCoNut [58] and Recoder [59]. Despite significant advancements, these methods remain limited by training data and struggle with unseen bugs [60].

The Emergence of LLM-based APR:

The recent growth of LLMs has transformed the APR field. LLM-based APR techniques demonstrate significant advancements over traditional state-of-the-art techniques, leveraging the advanced code-generation capabilities of modern LLMs [61]. Consequently, LLMs form the foundation of a new APR paradigm [18, 62].

Recent research has led to the emergence of several LLM-based APR paradigms, which can be grouped into four main categories:

- **RAG approaches** enhance bug repair by retrieving relevant context, such as code documentation stored in vector databases, during the repair process [2]. This approach allows access to external knowledge, enhancing LLMs' bug-fixing capabilities [1, 54].
- **Interactive/Conversational approaches** utilize LLMs' dialogue capabilities, providing instant developer feedback during patch validation [15, 16]. This iterative feedback loop refines generated patches to achieve better outcomes [15].
- **Agent-based approaches** enhance bug localization and repair by equipping LLMs with the ability to access external environments, operate tools (e.g. file editors, terminals, web search engines), and make autonomous decisions [62, 2, 63]. Using multi-step reasoning, these frameworks replicate developers' cognitive processes through specialized agents [17, 10, 8]. Examples include SWE-Agent [13], FixAgent [8], MarsCodeAgent [12], and GitHub Copilot [45].
- **Agentless approaches** focus on simplicity and efficiency, reducing complex multi-agent coordination while maintaining effectiveness [9, 2]. These approaches provide clear guardrails for LLMs, improving transparency. The three-step approach (localization, repair, validation) of Agentless approaches achieves promising results at low cost [9, 63].

Popular LLMs for APR include ChatGPT, Codex, CodeLlama, DeepSeek-Coder, and CodeT5 [1, 54, 62]. Despite the significant advancements brought by LLMs, state-of-the-art APR systems continue to face notable challenges and limitations. Existing systems are often complex, with limited transparency and control over the bug-fixing process [9, 2, 1]. Additionally, the repairs are computationally intensive and time-consuming, leading to high costs [14, 2]. Furthermore a barrier to practical adoption remains: most APR systems are developed and evaluated in controlled environments. Consequently, there is still a lack of research and experience on integrating these approaches into real-world software development workflows and projects [64, 2].

2.3.2. APR Benchmarks

To standardize the evaluation of new APR approaches, benchmarks have been developed. These benchmarks consist of software bugs and issues, along with their corresponding fixes or tests, which can be used to evaluate the effectiveness of different APR techniques [62]. They are essential for comparing the performance of different APR systems and for understanding their strengths and weaknesses [2]. APR benchmarks are available for various programming languages, with a selection of popular benchmarks listed in Table2.1 below [65].

Model	**Languages**	**Number of Bugs**	**Description**
QuixBugs [66]	Python, Java	40	small single line bugs
Defects4J [67]	Java	854	real-world Java bugs
ManyBugs [68]	C	185	real-world C bugs
SWE Bench [69]	Python	2294	Real GitHub repository defects
SWE Bench Lite [70]	Python	300	selected real GitHub defects

Table 2.1.: *Overview of APR benchmarks*

3. Method

The primary objective of this thesis is to implement and assess the potentials and limitations of integrating LLM-based automated bug fixing into CI. We aim to answer the following research questions to evaluate the feasibility, capabilities and impact on the software development process of this approach.

- **RQ1:** How can LLM-based automated bug fixing be effectively and efficiently integrated into a CI pipeline?
- **RQ2:** What are the potentials and limitations of this integrated approach with respect to repair success rate, execution times and cost-effectiveness?

To answer these questions, we structured the process into three phases: preparation, implementation/usage, and evaluation. The process is visualized below in Figure 3.1.

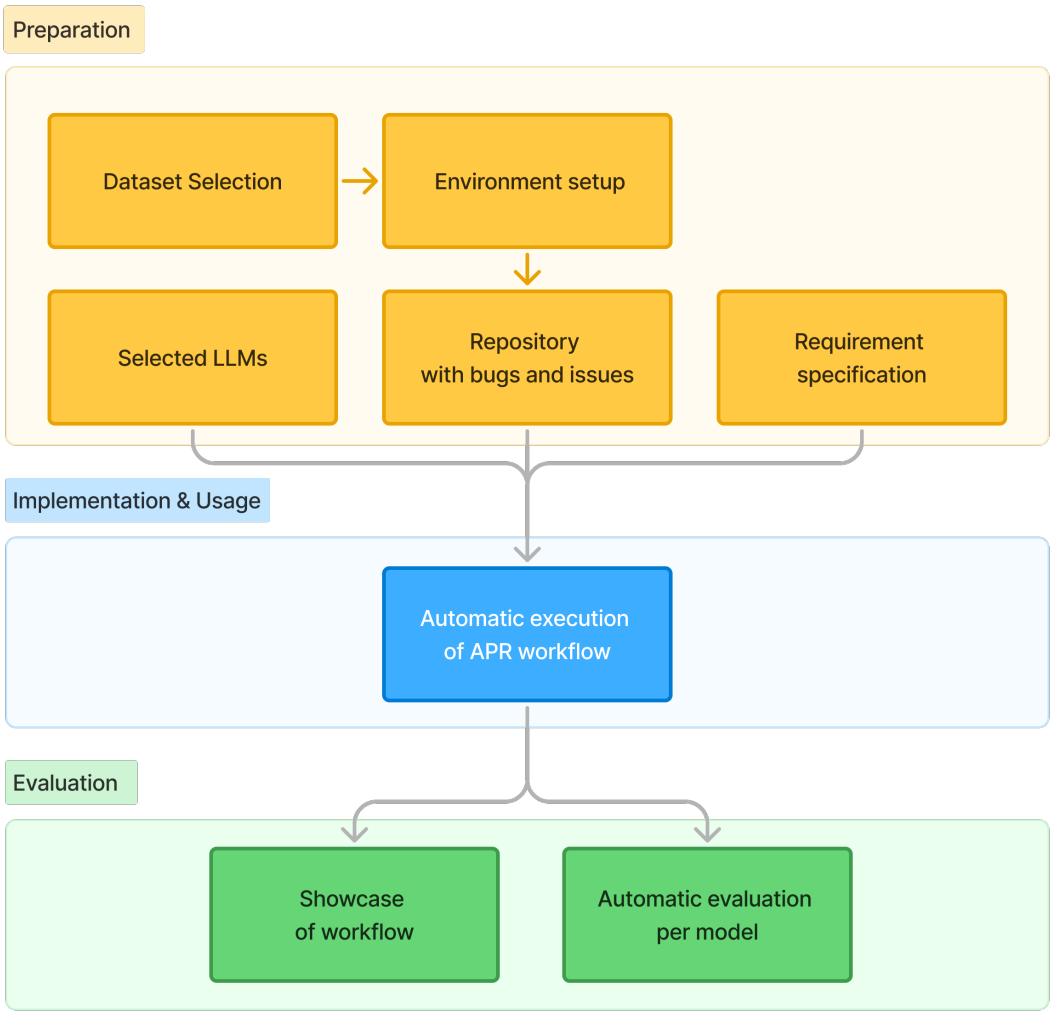

Figure 3.1.: *Thesis methodology approach, Source: own representation*

In the preparation phase, we select a suitable APR benchmark and a pool of LLMs. With the benchmark, we set up a realistic development environment on GitHub. By specifying requirements, we lay the groundwork for the implementation of the APR system. In the second part, we implement the APR system based on the requirements. With this implementation we evaluate the self-developed prototype using the evaluation metrics (listed in Section 3.3) collected during and after the execution of the APR system. Furthermore, we showcase the resulting workflow of using the system in the prepared repository. The following sections will go into detail about each of these phases.

3.1. Preparation

For implementing and evaluating our system, we prepared an environment where the system can be integrated, used, and evaluated. This includes selecting a suitable dataset and LLMs, setting up the environment, and specifying the requirements for the system.

3.1.1. Dataset Selection

For evaluating the effectiveness of our APR integration, we selected the QuixBugs benchmark [66]. This dataset is well-suited for our purposes due to its focus on small-scale bugs in Python[3]. It consists of 40 individual files, each containing an algorithmic bug caused by a single erroneous line. Corresponding tests and a corrected version for every file are also included in the benchmark, which allows for seamless repair validation. QuixBugs was developed as a set of challenging problems for developers [66], enabling us to evaluate whether our system can take over the cognitively demanding task of fixing small bugs without developer intervention. Compared to other APR benchmarks 2.1 like SWE-Bench [69], QuixBugs is relatively small, which allows for accelerated setup and development.

3.1.2. Large Language Model Selection

For the evaluation of our APR system, we test a selected pool of LLMs. We evaluate with the latest models[4] from the three vendors that currently dominate AI-assisted coding workflows: Google, OpenAI, and Anthropic. The models are selected to cover a range of capabilities and costs, with a focus on lower-tier models, allowing us to evaluate the performance, execution times and cost-effectiveness of the APR system. Table 3.1 shows twelve selected models that are used for evaluation with the following data:

- (1) Model Name: The name of the LLM.
- (1a) Abbreviation: Short identifier used for results and tables.
- (2) Context Window Size in Tokens: The maximum number of tokens the model can process in a single request.

[3]QuixBugs also includes translated Java versions, which are excluded from our evaluation.
[4]released before 11 July 2025

- (3) Cost per 1M Tokens: The cost of processing 1 million tokens, divided into input and output cost.
- (4) Provider Description: Description of the model's characteristics.
- (5) Source: The source where the model information was obtained.

Table 3.1.: *Characteristics of selected LLMs*

(1)	(1a)	(2)	(3)	(4)	(5)
gemini-2.0-flash-lite	G2F-L	1M	input: $0.075 output: $0.30	Cost efficiency and low latency	[71]
gemini-2.0-flash	G2F	1M	input: $0.15 output: $0.60	Next generation features, speed, and real-time streaming	[71]
gemini-2.5-flash-lite	G25F-L	1M	input: $0.10 output: $0.40	Most cost-efficient model supporting high throughput	[71]
gemini-2.5-flash	G25F	1M	input: $0.30 output: $2.50	Adaptive thinking, cost efficiency	[71]
gemini-2.5-pro	G25P	1M	input: $1.25 output: $10.00	Enhanced thinking and reasoning, multimodal understanding, advanced coding, and more	[71]
gpt-4.1-nano	GPT4N	1M	input: $0.10 output: $0.40	Fastest, most cost-effective	[72]
gpt-4.1-mini	GPT4M	1M	input: $0.40 output: $1.60	Balanced for intelligence, speed, and cost	[72]
gpt-4.1	GPT4	1M	input: $2.00 output: $8.00	Flagship GPT model for complex tasks	[72]
o4-mini	O4M	200k	input: $1.10 output: $4.40	Optimized for fast, effective reasoning with exceptionally efficient performance in coding	[72]
claude-3-5-haiku	C35H	200k	input: $0.80 output: $4.00	Our fastest model	[73]
claude-3-7-sonnet	C37S	200k	input: $3.00 output: $15.00	High-performance model with early extended thinking	[73]
claude-sonnet-4-0	CS40	200k	input: $3.00 output: $15.00	High-performance model	[73]

The exact snapshot versions of each model used are listed in the Appendix A.

3.1.3. Environment Setup

To mirror a realistic software development environment, we prepared a GitHub repository containing the QuixBugs Python dataset. This repository serves as the basis for the bug-fixing process, allowing the system to interact with the codebase and perform repairs.

Using the 40 buggy files, we generate a GitHub issue for each. A consistent issue template is used, capturing only the title of the problem with a minimal description. The generated issues serve as the entry point and communication medium for the APR system. Figure 2.3 shows an example of a generated GitHub issue.

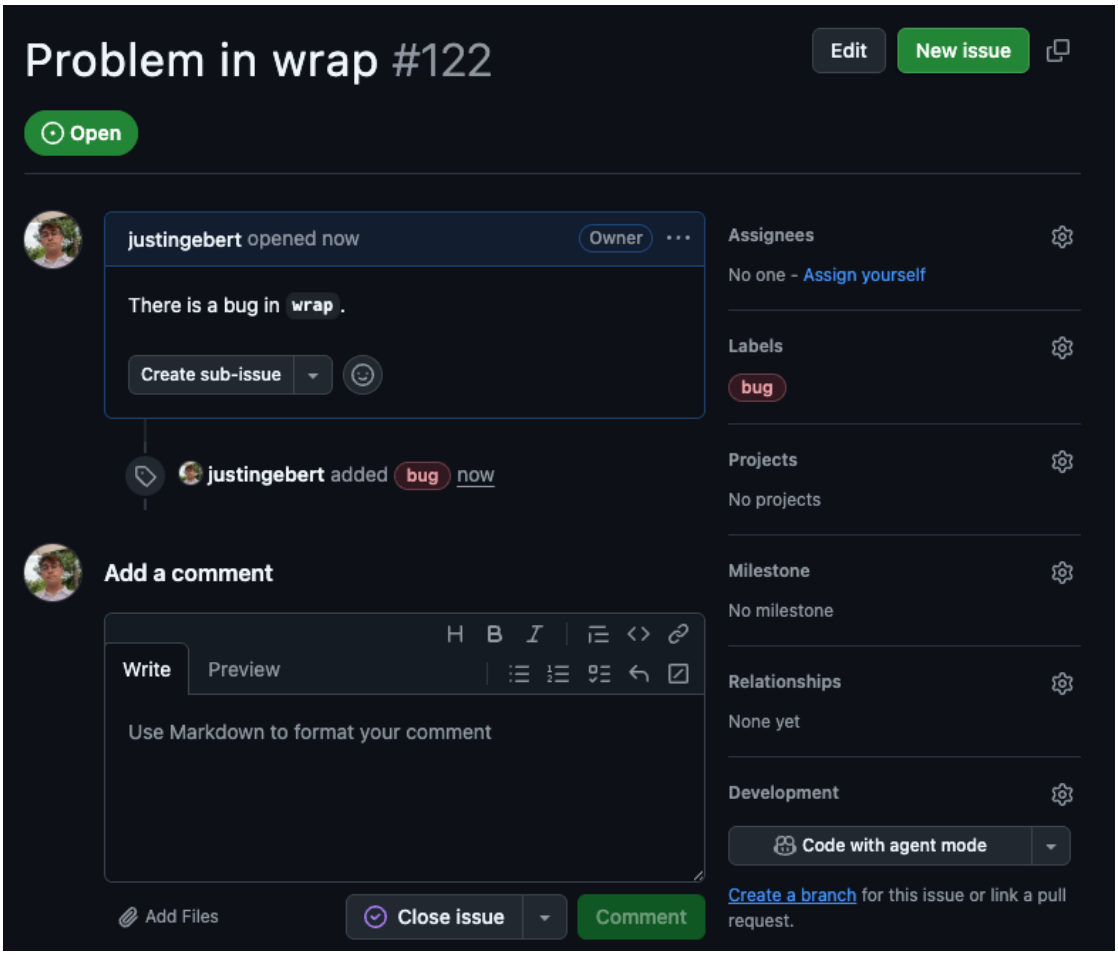

Figure 3.2.: *Example of a generated GitHub Issue, Source: screenshot*

3.1.4. Requirements Specification

Before the implementation phase, we constructed requirements for the APR prototype. We followed the INVEST model, a widely adopted method in Agile software development for engineering requirements [74]. According to the INVEST principles, each requirement was formulated to be independent, negotiable, valuable, estimable, small, and testable. Using this framework, we defined both functional and non-functional

requirements. The requirements are precisely defined, verifiable, and easily adaptable to iterative development. Chapter 4 showcases the resulting requirements.

3.2. System Implementation

In this section, we provide a high-level overview of the implemented automated bug fixing CI pipeline. Chapter 5 provides a more detailed description of the implementation.

The prototype was developed using iterative prototyping and testing, with a focus on simplicity and extensibility. It is based on the self-developed requirements. An overview of the system is visualized in Figure 3.3 and Figure 3.4.

Figure 3.3 shows that once the system is in place, a CI pipeline can be triggered by different events. This executes the CI pipeline on a GitHub Action runner. The first pipeline job fetches and filters relevant issues. Resulting issues are passed to the APR Core, which contains the main bug-fixing logic. The APR Core (visualized in Figure 3.4) runs in a container and communicates with the configured LLMs via API to localize and fix the issue. With the generated file edits, the changes are validated and tested. When validation passes, the changes are applied and a pull request is opened on the repository, linking the issue. In case of an unsuccessful repair or exhaustion of the maximum number of attempts, a failure is reported to the issue.

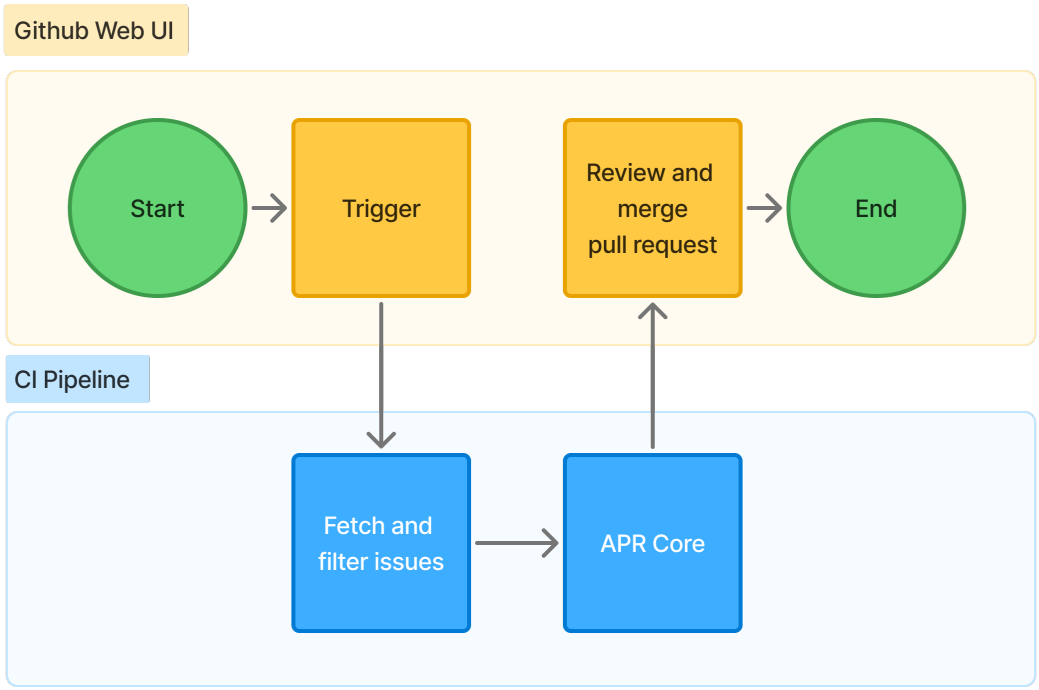

Figure 3.3.: *Overview of the CI Pipeline, Source: own representation*

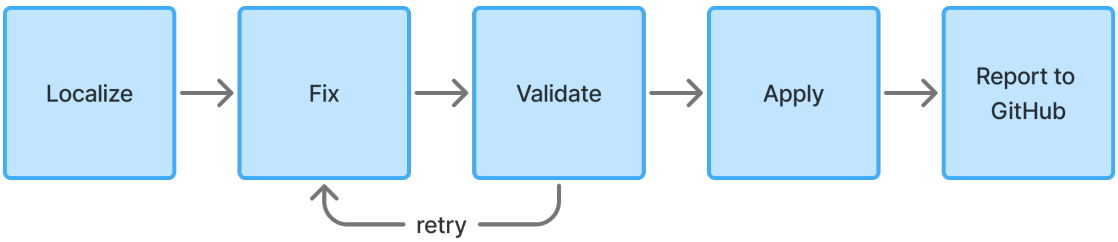

Figure 3.4.: *Overview of the APR Core, Source: own representation*

3.3. Evaluation

This section describes how we measure the effectiveness, performance and cost of the APR system when integrated into a GitHub repository. We use GitHub Actions' CI capabilities in combination with the prepared QuixBugs repository as a base. All twelve selected LLM models from Table 3.1.2 are evaluated using the prototype. The evaluation is based on data collected during and after a execution of the APR system.

We focus on several key data metrics to assess the system's performance and capabilities in repairing software bugs. These metrics provide insights into effectiveness, performance, reliability, cost and overall impact on the SDLC.

Effectiveness of the LLMs and the approach, is measured by the repair success rate. A repair is considered successful if it passes both the validation stage and the complete test suite provided by QuixBugs. Specifically, an issue is deemed fixed when the generated code is syntactically correct and all associated tests pass without errors.

To assess performance, we analyze collect timings of issue repair attempts. Feasibility is evaluated by estimating the cost of a repair attempt based on the number of tokens and token pricing listed by API providers. GitHub Action minutes are not included in the cost estimation, as 2000 minutes are included in the free tier of GitHub for public repositories [75].

Furthermore, we evaluate whether multiple attempts[5] can help improve the repair success rate and how this relates to the execution time and cost of the repair process.

Each execution of the APR system collects repair process data. Using self-developed Python scripts, we aggregate the data from different sources for each run. Below, we summarize data collected for each run (Table3.2), each issue processed (Table3.3), and each stage of an issue repair process (Table 3.4). We use this data to calculate results for runs testing different configurations with each of the selected LLMs, which will help in answering RQ2. Table 3.5 shows the resulting calculations.

[5]equivalent to few shot prompting

Table 3.2.: *Summary of metrics collected for each run*

Metric	Description	Source
RunID	Unique identifier for the workflow run	Github Action Runner
Configuration	Configuration details, including LLM and maximum attempts allowed	Github repository & APR Core
Execution Times	Total time taken for the run	GitHub API & APR Core
Job Execution Times	Time taken for each job in the pipeline	Github API
Issues Processed	Data of issues processed shown in Table 3.3	APR Core

Table 3.3.: *Summary of metrics collected for each processed issue*

Metric	Description	Source
IssueID	Unique identifier of the issue	Github Issue ID
Repair Successful	Boolean indicating whether the repair was successful	APR Core
Number of Attempts	Total attempts made	APR Core
Execution Time	Time taken to process the issue, including all stages	APR Core
Tokens Used	Number of tokens processed by the LLM during the repair process	LLM API
Cost	Cost associated with the repair process, calculated based on tokens * cost per token	APR Core
Stage Information	Details about each stage of the repair process shown in Table 3.4	APR Core

Table 3.4.: *Summary of metrics collected for each stage*

Metric	Description	Source
StageID	Unique identifier of the stage	APR Core
Stage Execution Time	Time taken for stage to complete	APR Core
Stage Outcome	Outcome of each stage, indicating success, warning or failure	APR Core
Stage Details	Additional details, such as error or warnings messages	APR Core

Table 3.5.: *Run evaluation metrics*

Metrics	**Calculation**
Repair Success Rate	$\frac{\textit{Number of Successful Repairs}}{\textit{Total Issues Processed}}$
Average Execution Time per Issue	$\frac{\textit{Total Execution Time}}{\textit{Total Issues Processed}}$
Average Cost per Issue	$\frac{\textit{Total Cost}}{\textit{Total Issues Processed}}$
APR Core Execution Time	*APR Core Execution Time*
CI Pipeline Overhead	*CI Pipeline Execution Time* − *APR Core Execution Time*

4. Requirements

To guide the implementation of the prototype, we developed the following requirements to help design and develop the prototype. These requirements enable structured planning and prioritization during development and track progress throughout the implementation. Both functional (Table4.1) and non-functional (Table4.2) requirements are listed below.

4.1. Functional Requirements

Table 4.1.: *Functional requirements*

ID	Title	Description	Verification
F0	Multiple Triggers	The Pipeline can be triggered: manually, scheduled or by issue labeling in GitHub.	Runs can be found for these triggers
F1	Issue Filtering	The system retrieves and filters GitHub issues based on the configured label.	Logs show list of passing issues.
F2	Code Checkout	The pipeline fetches the repository code into a fresh workspace and branch.	Checkout and branch visible in logs.
F3	Bug Localization	The LLM analyzes the issue description and identifies the file(s) needing changes.	LLM output contains file paths.
F4	Fix Generation	The LLM generates and proposes code edits for the identified files.	LLM output contains adjusted content.
F5	Change Validation	All generated changes are validated via formatting, linting, and running relevant tests.	Logs show results of format/lint/test steps.
F6	Iterative Patch Generation	If F5 reports failures, retry F4-F5 until maximum number of attempts is reached.	Logs show retries with multiple stage executions.
F7	Patch Application	Commit LLM-generated edits to the issue branch.	Git branch and commit with reference to the issue is visible.

F8	Result Reporting	The system reports the outcome by creating a Pull Request or adding a comment to the issue.	Pull Requests and issue comments are visible.
F9	Log and Metric Collection	For every run, logs and key metrics are collected and made available.	Artifacts and console logs are available in CI run results.

4.2. Non-Functional Requirements

Table 4.2.: *Non-Functional requirements*

ID	Title	Description	Verification
N0	Container Runtime	All core logic runs inside a Docker container in the CI environment.	CI logs confirm use of Docker container.
N1	Configurable	Users can configure labels, branches, number of attempts and LLM in a YAML file.	The system loads a custom configuration.
N2	Portable	The system can be deployed and run on any Python GitHub repository with minimal setup.	Demonstrate successful run on at least two different repositories.
N3	Run Repeatable	Runs are deterministic given identical repo state and config.	Multiple runs on the same issue report similar metrics.
N4	Observable	Logs and metrics are generated for every run and accessible as CI artifacts.	Downloadable artifacts and visible logs for each run.
N5	End-to-End Automation	The process is fully automated from issue creation to PR no manual intervention is needed.	No manual steps are needed, automation is shown.

5. Implementation

In this section we break down the implementation of the system into its core components, following the methodology and requirements outlined in the previous chapters. The complete implementation and source code be found in the appendix A.

The resulting prototype consists of two main components. The **APR Core** which holds the core logic for the repair process and a **CI pipeline** which integrates the APR core logic within a GitHub repository. The **CI pipeline!** (**CI pipeline!**) serves as an entry point and orchestrates the execution of the APR Core based on configured trigger events in the repository.

5.1. System Components

The implementation of the main components will be described in detail in the following section.

APR Core:

The APR Core contains the main bug fixing logic written in Python. Embedding it into a Docker Image[6] makes it easy to deploy and portable. In order to use the APR Core the following data (displayed in Table 5.1) needs to passed to the container:

Table 5.1.: *APR Core container inputs*

Name	Description	Type
Source Code	Git repository where to repair bugs in	Volume mount
GITHUB_TOKEN	Token for GitHub API authentication	Environment variable
LLM_API_KEY	API key for the LLM provider	Environment variable
ISSUE_TO_PROCESS	The issue to process in JSON format	Environment variable
GITHUB_REPOSITORY	GitHub repository for fetching and writing data	Environment variable

With this environment set, the APR Core iterates over all issues fetched from the "ISSUE_TO_PROCESS" environment variable. For each issue, the main APR logic is

[6] A Docker image is a standardized package that contains all code, libraries and configuration required to run a containerized application [76].

executed. This logic follows a predefined flow that makes use of multiple stages and tools.

First, a clean workspace and the issue repair context is set up. The context acts as the central data structure for the issue repair process and is used at every step. Listing 5.1 shows what the context looks like when initialized.

```
context = {
    "bug": issue,
    "config": config,
    "state": {
        "current_stage": None,
        "current_attempt": 0,
        "branch": None,
        "repair_successful": False,
    },
    "files": {
        "source_files": [],
        "fixed_files": [],
        "diff_file": None,
        "log_dir": str(log_dir),
    },
    "stages": {},
    "attempts": [],
    "metrics": {
        "github_run_id": os.getenv("GITHUB_RUN_ID"),
        "script_execution_time": 0.0,
        "execution_repair_stages" : {},
        "tokens": {}
    },
}
```

Listing 5.1: *Context JSON*

This context object is passed between stages, with each stage performing a specific task in the bug fixing process and returning an updated context. The APR core uses four implemented stages: Localize, Fix, Build, and Test.

The repair process for an issue starts with the localization stage. This stage attempts to identify the files in the codebase required to fix the bug, using the configured LLM via the provider's Software Development Kit (SDK). A localization prompt is built using the issue and a constructed hierarchy of the repository's file structure. The response is expected to return a list of files where the bug might be located. System instruction and prompt for localization are show below in Listing 5.2.

```
system_instruction = "You are a bug localization system. Look at the
    issue description and return ONLY the exact file paths that need to be
     modified."

prompt = f"""
    Given the following GitHub issue and repository structure, identify
    the file(s) that need to be modified to fix the issue.

    Issue #{issue['number']}: {issue['title']}
    Description: {issue.get('body', 'No description provided')}

    Repository files:
    {json.dumps(repo_files, indent=2)}

    Return a JSON array containing ONLY the paths of files that need to
    be modified to fix this issue.
    Example: ["path/to/file1.py", "path/to/file2.py"]
    """
```

Listing 5.2: *Localization Prompt*

With localized files in the context, the Fix stage comes next. This stage calls the configured LLM API to generate a fix for the issue in the localized files. To construct the prompt, issue details and relevant file names, along with their content, are included. Responses should specify the necessary edits for each file, or indicate when no changes are required. After parsing the generated output, edits are applied to the corresponding files in the workspace. The context is then updated to reflect the new file content. Listing 5.3 shows the system instruction and base prompt used for the Fix stage.

```
system_instruction = "You are part of an automated bug-fixing system.
    Please return the complete, corrected raw source files for each file
    that needs changes, never use any markdown formatting. Follow the
    exact format requested."

base_prompt = f"""
    The following Python code files have a bug. Please fix the bug across
     all files as needed.

    {files_text}

    Please provide the complete, corrected source files. If a file doesn'
    t need changes, you can indicate that.
    For each file that needs changes, provide the complete corrected file
     content.
    Format your response as:

    === File: [filepath] ===
    [complete file content or "NO CHANGES NEEDED"]

    === File: [filepath] ===
    [complete file content or "NO CHANGES NEEDED"]
    """
```

Listing 5.3: *Repair Prompt*

For validating generated fixes, two stages are used: Build and Test. The Build stage is responsible for checking if the code can built by validating the syntax the changes made in the Fix stage. For Python code, this means checking if all syntax is valid and follows standardized code quality and maintenance rules. To achieve this, the code is first formatted using the Python formatter Black[7] and then linted using flake8[8]. This ensures properly formatted code and appends any warnings or errors to the context.

If a test command is configured, the next stage of the validation process tests the generated changes. The Test stage runs the tests defined in the repository using the configured test command for each fixed file. In case tests fail, the context is updated with the error messages, and the repair returns to the Fix stage for a new attempt.

For a new attempt, additional feedback is generated using the previous code and stage results. This feedback gets attached to the prompt. When reaching the maximum number of attempts without passing tests, an unsuccessful repair is reported by creating a comment on the issue using the GitHub API.

If both validation stages return a success, the issue is marked as successfully repaired. The file changes are committed and pushed to the remote repository on GitHub. A pull request is then created to merge the issue branch into the main branch. This pull request includes detailed file diffs and links the associated issue.

At this point, integration with the GitHub repository is completed. During execution, every action of the APR Core is logged, logs can be used for debugging and result in a more transparent repair process. Furthermore, it collects metrics such as the number of attempts, execution times, and token usage, which are essential for analyzing the effectiveness and performance of the APR approach. A summary of the metrics collected is mentioned in Section 3.3.

The APR core is designed to be modular and extensible, allowing for future enhancements and additional stages or tools to be integrated as needed. It is also designed to be lightweight, ensuring that it can run efficiently within a CI environment. Figure 5.1 illustrates the previously explained functionalities (blue) and tools (green) used in the APR Core.

[7] Black is a code formatter for Python that reformats code to comply with its style guide, aiming for consistent and readable code [77].

[8] Flake8 tool that checks Python source code for compliance with coding standards and potential errors [78].

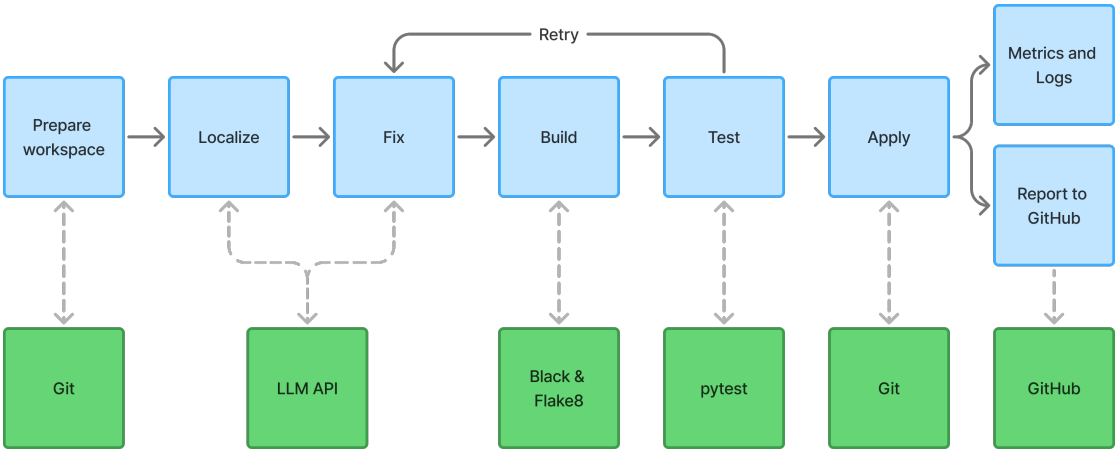

Figure 5.1.: *APR Core, Source: own representation*

CI Pipeline:

The APR Core is integrated into GitHub using a GitHub Action workflow to create a CI pipeline. This workflow is written in YAML according to the GitHub Action standard [79]. The pipeline is executed, on a Linux x64 runner provided and hosted by GitHub. This eliminates the overhead of managing our own runners but comes at the cost of uncertain performance and availability.

The workflow is made up of four triggers and three jobs. Triggers are based on events[9] from the GitHub repository and serve as the entry point for executing the jobs. Given the triggers, the workflow can be executed in two different ways:

- Batch processing by fetching all issues marked for repair. This can be triggered by a manual dispatch ("workflow_dispatch") or scheduled execution ("cron").
- Processing a single issue from an event. The issue is passed from the ("issue_labeled") event when labeled with the configured label and from (‘"issue_comment") when information is added to the issue in the form of a comment.

The trigger event information gets passed as environment variables to the first job named "gate". This job uses the event data to determine whether the issue should be processed or skipped. This is determined by a Python script ("filter_issues.py"). This script must be placed accessible to the workflow file. It checks the labels of issues and evaluates their state to determine which are relevant for the APR process.

If no issues pass the "gate", the job "skipped" is executed, which logs that no issues were found and exits the workflow run.
If at least one issue passes the "gate", the "bugfix" job is started. This job is responsible for executing the APR Core logic. It sets up the necessary prerequisites (see Table 5.1) to start a container using the latest version of the APR Core Docker Image, which performs the repair. These include checking out and mounting the code repository, setting environment variables, and providing the necessary permissions for the APR Core to edit repository content, create pull requests, and write to issues on GitHub.

[9]Events are specific activities or changes in a repository. [80]

The final step of "bugfix" uploads all logs and metric files from the APR Core as artifacts to GitHub, making them available after the workflow run has completed.

Figure 5.2 visualizes and overview of the the workflow and its functionality.

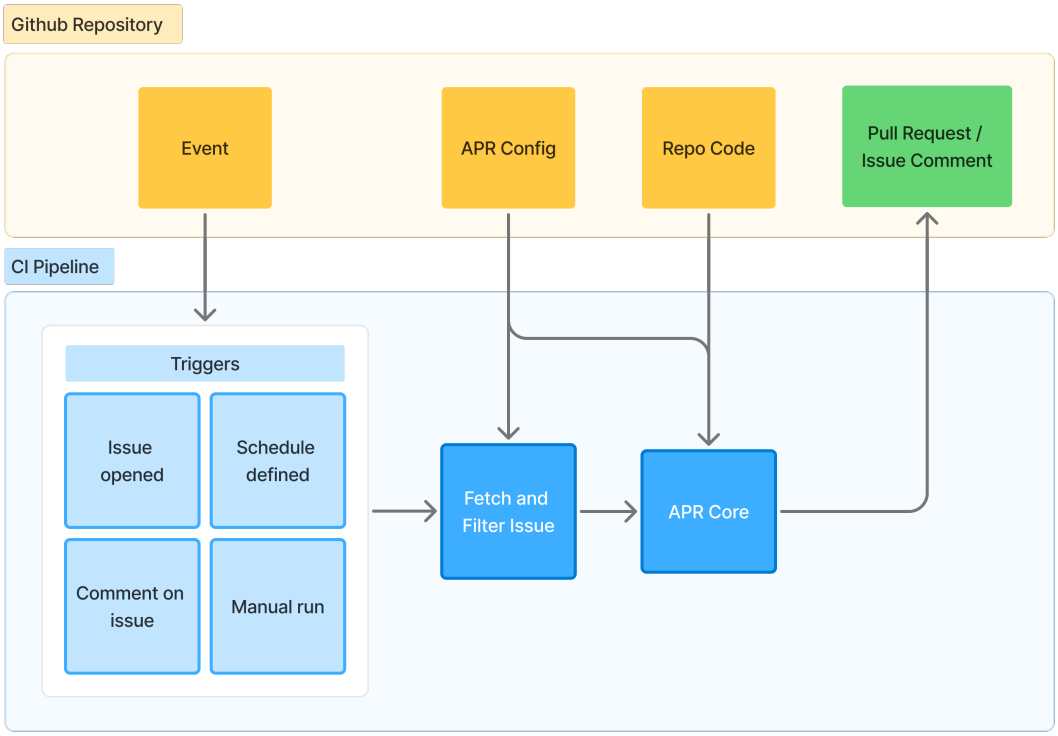

Figure 5.2.: *CI Pipeline, Source: screenshot*

To use this integration in a repository, the workflow file must be placed in the ".github/-workflows" directory of the repository along with "filter_issues.py" in ".github/scripts". With this in place, an "LLM_API_KEY" must be set as a secret in the repository settings. This key is used by the APR Core to authenticate with the configured LLM provider API. Lastly, GitHub Actions must be granted permissions to create pull requests and write to issues in the repository. This is done by setting the workflow permissions in the repository settings under Actions > General > Workflow permissions. Detailed setup instructions can be found in the repository listed in the Appendix A

5.2. System Configuration

To make the system easily adjustable, the APR Core and the CI pipeline can be configured using a YAML configuration file. This configuration is optional, when no configuration is provided, the system falls back to a default setup. A custom configuration must be named "bugfix.yml" and placed at the root of the repository. A placed configuration is read by system components during execution. This setup allows

for easy customization of the systems behavior without modifying the underlying code. Table 5.2 lists the available configuration fields along with their descriptions.

Table 5.2.: *Configuration Fields and Descriptions*

Configuration Field	Description
to_fix_label	The label used to identify issues that need fixing.
submitted_fix_label	The label applied to issues when a fix is submitted.
failed_fix_label	The label applied to issues when a fix failed.
workdir	The working directory where the code lives.
test_cmd	The command used to run tests on the codebase.
branch_prefix	The prefix for branches created for bug fixes.
main_branch	The main branch where bug fix branches are based.
max_issues	The maximum number of issues to process in a single run.
max_attempts	The maximum number of attempts to fix an issue.
provider	The LLM provider used for generating fixes.
model	The specific model from the LLM provider.

5.3. Requirement Validation

This section demonstrates how the prototype satisfies the functional and non-functional requirements defined in Chapter 4. Table 5.3 summarizes the satisfaction and verification of the requirements with provided screenshots, context excerpts and log excerpts captured during development. Entire logs, configuration files, and context JSON files are available in the GitHub repositories listed in Appendix A.

Table 5.3.: *Requirement Satisfaction and Validation*

ID	Title	Satisfied	Verification / Reference
F0	Multiple Triggers	Yes	See Figure 5.3
F1	Issue Filtering	Yes	See filtered issues log in Listing 5.4
F2	Code Checkout	Yes	See log excerpt in Listing 5.5
F3	Bug Localization	Yes	See log excerpt in Listing 5.6
F4	Fix Generation	Yes	See context excerpt in Listing 5.7

F5	Change Validation	Yes	See log excerpt in Listing 5.8
F6	Iterative Patch Generation	Yes	See log excerpt in Listing 5.9
F7	Patch Application	Yes	See log excerpt in Listing 5.10
F8	Result Reporting	Yes	See Figure 5.4 and Figure 5.5
F9	Log and Metric Collection	Yes	See Figure 5.6
N1	Container Runtime	Yes	See log excerpt in Listing 5.11
N2	Configurable	Yes	See log excerpts from Listing 5.12
N3	Portable	Yes	Used in two repositories: see Appendix A
N4	Run Repeatable	No	Due to non deterministic LLM outputs and varying execution environments, identical runs resulted in different results. See discussion in Section 7.1.
N5	Observable	Yes	See Figure 5.6fig:metrics.
N6	End-to-End Automation	Yes	See full process in Section 6.1

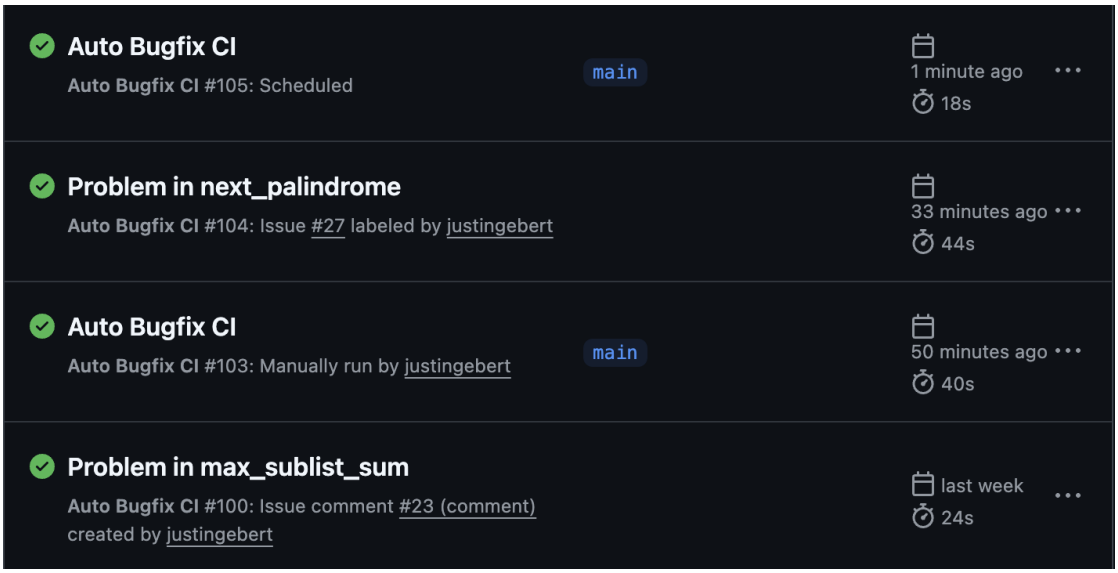

Figure 5.3.: *Workflow triggers for APR system, Source: screenshot*

```
Found 1 issues to process
[{"number": 27, "title": "Problem in next_palindrome", "body": "There is a bug
    in **‘next_palindrome‘**.", "labels": ["bug_v01", "quixbugs"]}]
```

Listing 5.4: *Filtered issues log excerpt*

```
Run actions/checkout@v4
Syncing repository: justingebert/bugfix-ci
Getting Git version info
Temporarily overriding HOME='/home/runner/work/_temp/6ca8e386-1931-4130-8b38-
    cd4993a6fa1f' before making global git config changes
Adding repository directory to the temporary git global config as a safe
    directory
/usr/bin/git config --global --add safe.directory /home/runner/work/bugfix-ci/
    bugfix-ci
Deleting the contents of '/home/runner/work/bugfix-ci/bugfix-ci'
Initializing the repository
Disabling automatic garbage collection
Setting up auth
Fetching the repository
Determining the checkout info
/usr/bin/git sparse-checkout disable
/usr/bin/git config --local --unset-all extensions.worktreeConfig
Checking out the ref
/usr/bin/git log -1 --format=%H
465a84dc8a989cd1ba7603324687f7d57cdc0998
```

Listing 5.5: *Code checkout log excerpt*

```
root - INFO - [localize] LLM response: ["next_palindrome.py"]
```

Listing 5.6: *Bug localization log excerpt*


```
    "fix_attempt_1": {
        "results": {
          "status": "success",
          "message": "Successfully fixed files",
          "details": {
            "fixed_files": [
              "/workspace/quixbugs/python_programs/next_palindrome.py"
            ],
            "tokens": {
              "input_tokens": 431,
              "output_tokens": 288,
              "total_tokens": 719,
              "cost": 0.0006332
            },
            "raw_response": "=== File: /workspace/quixbugs/
    python_programs/next_palindrome.py ===\ndef next_palindrome(digit_list
    ):\n    high_mid = len(digit_list) // 2\n    low_mid = (len(digit_list
    ) - 1) // 2\n    while high_mid < len(digit_list) and low_mid >= 0:\n
            if digit_list[high_mid] == 9:\n            digit_list[high_mid]
     = 0\n            digit_list[low_mid] = 0\n            high_mid += 1\n
```

```
            low_mid -= 1\n        else:\n            digit_list[
high_mid] += 1\n            if low_mid != high_mid:\n
digit_list[low_mid] = digit_list[high_mid]\n            return
digit_list\n    return [1] + (len(digit_list) - 1) * [0] + [1]\n\n\"\"
\"\nFinds the next palindromic integer when given the current integer\
nIntegers are stored as arrays of base 10 digits from most significant
 to least significant\n\nInput:\n    digit_list: An array representing
 the current palindrome\n\nOutput:\n    An array which represents the
next palindrome\n\nPreconditions:\n    The initial input array
represents a palindrome\n\nExample\n    >>> next_palindrome
([1,4,9,4,1])\n    [1,5,0,5,1]\n\"\"\""
      }
    },
    "duration": 7.0682
  },
```

Listing 5.7: *Fix generation log excerpt*

```
root - INFO - == Running stage: build ==
root - INFO - [build] starting build process
root - INFO - [build] Formatting /workspace/quixbugs/python_programs/next_palindrome.
py with black...
root - INFO - [build] /workspace/quixbugs/python_programs/next_palindrome.py
formatted successfully by black.
root - INFO - [build] Linting /workspace/quixbugs/python_programs/next_palindrome.py
 with flake8...
root - WARNING - [build] flake8 found issues in /workspace/quixbugs/python_programs/
next_palindrome.py:
/workspace/quixbugs/python_programs/next_palindrome.py:20:80: E501 line too long (90
 > 79 characters)

root - ERROR - [build] Build failed. 1 files with issues.
root - INFO - == Stage build completed in 0.5411 seconds ==
root - INFO - == Running stage: test ==
root - INFO - [test] running tests for fixed files
root - INFO - [test] Testing next_palindrome...
root - INFO - [test] Looking for specific test: /workspace/quixbugs/python_testcases
/test_next_palindrome.py
root - INFO - [test] Executing test command: cd /workspace/quixbugs && python -m
pytest python_testcases/test_next_palindrome.py -v
root - INFO - [test] Specific test passed for next_palindrome.
```

Listing 5.8: *Change validation log excerpt*

```
root - INFO - [test] Specific test failed for wrap.
root - ERROR - [test] Tests failed. 1 files with test failures.
root - INFO - == Stage test completed in 0.5417 seconds ==
root - INFO - === Repair failed on attempt 1/3, trying again ===
root - INFO - Resetting to main branch
root - INFO - Reset /workspace/python_programs/wrap.py to main branch
root - INFO - === Attempt 2/3 for issue #47 ===
root - INFO - == Running stage: fix ==
root - INFO - Loaded file: /workspace/python_programs/wrap.py
```

Listing 5.9: *Iterative patch generation log excerpt*

```
root - INFO - === Repair successful on attempt 1/1 ===
root - INFO - Added file /workspace/quixbugs/python_programs/next_palindrome.py to
staging area
root - INFO - 1 files with changes committed to branch bugfix_v01_27
root - INFO - Push successful
```

Listing 5.10: *Branch and commit created for bugfix*

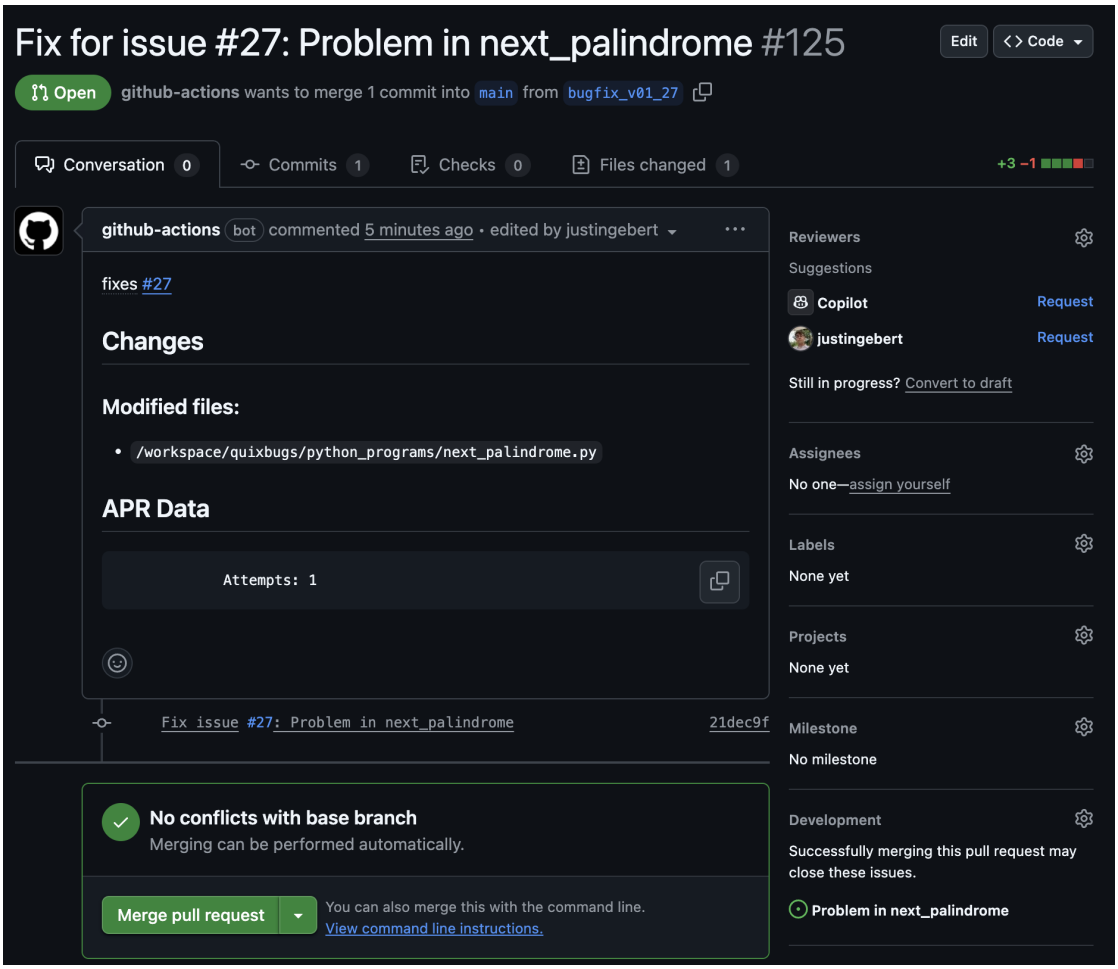

Figure 5.4.: *Pull request report for bugfix, Source: screenshot, Source: screenshot*

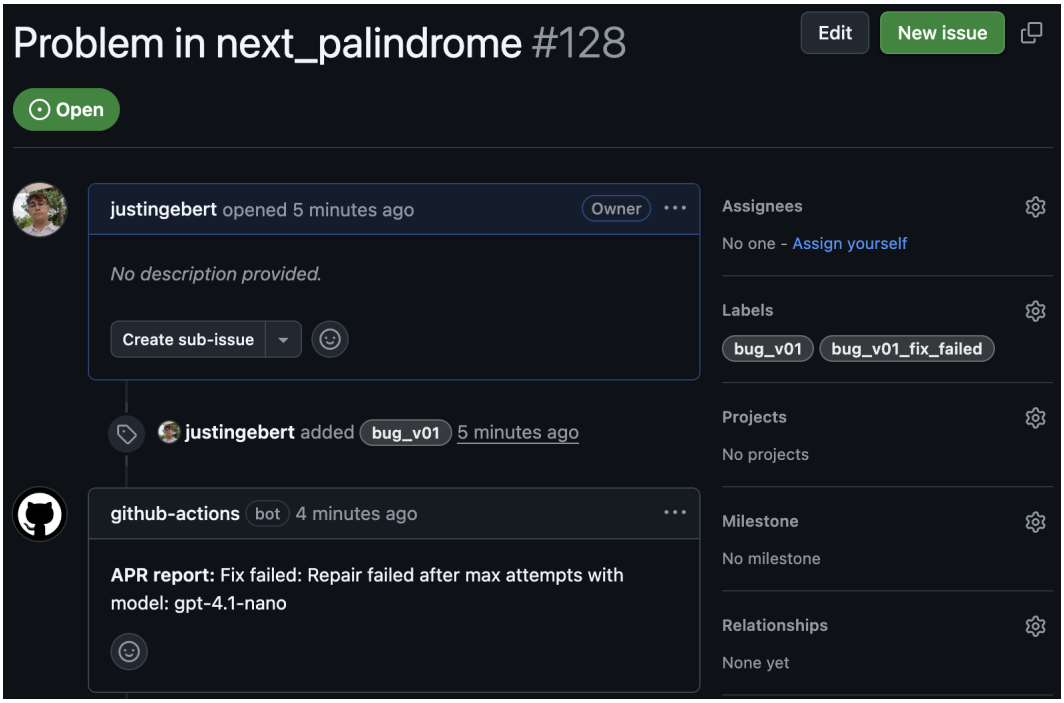

Figure 5.5.: *Issue comment reporting repair result, Source: screenshot*

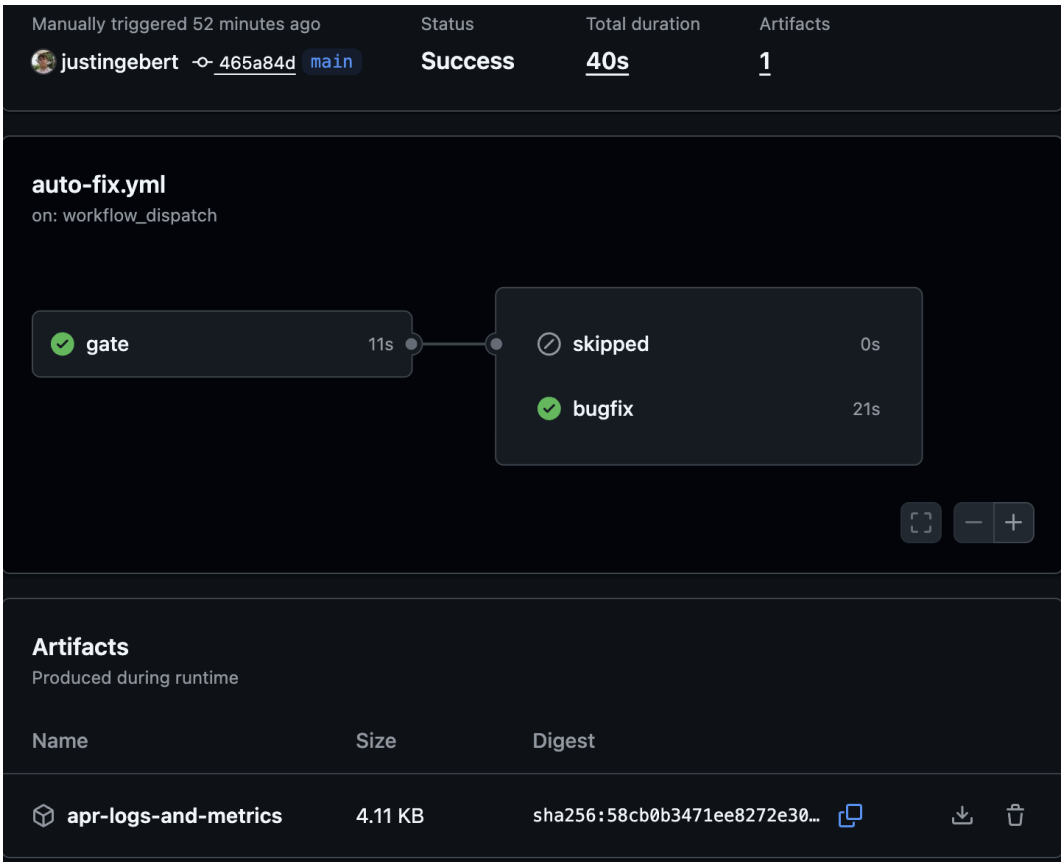

Figure 5.6.: *Downloadable Artifacts from workflow run, Source: screenshot*

```
Downloaded newer image for ghcr.io/justingebert/bugfix-ci:latest
Run docker run --rm \
docker run --rm \
  -v "$GITHUB_WORKSPACE:/workspace" \
  -e GITHUB_TOKEN -e GITHUB_RUN_ID -e GITHUB_REPOSITORY -e LLM_API_KEY -e
FILTERED_ISSUES \
  ghcr.io/justingebert/bugfix-ci:latest
shell: /usr/bin/bash -e {0}
env:
  GITHUB_TOKEN: ***
  GITHUB_REPOSITORY: justingebert/bugfix-ci
  GITHUB_RUN_ID: 16531173399
  LLM_API_KEY: ***
  FILTERED_ISSUES: [{"number": 27, "title": "Problem in next_palindrome", "body": "
There is a bug in **‘next_palindrome‘**.\n\n* **File**: ‘/Users/justingebert/
Projects/Uni/BA/bugfix-ci/datasets/quixbugs/python_programs/next_palindrome.py‘", "
labels": ["bug_v01", "quixbugs"]}]
```

Listing 5.11: *Docker container runtime log excerpt*

```
root - INFO - [info] loaded default config from apr_core/default-config.yml
root - INFO - [info] loaded config from /workspace/bugfix.yml
```

Listing 5.12: *Load custom configuration*

6. Results

In the following section we present the results of our implementation and evaluation. We implemented a working prototype for assessing how to integrate LLM-based Automated Bug Fixing into CI, and the resulting potentials and limitations of using this system in software development workflows.

The setup and usage of the prototype in a GitHub repository is demonstrated in the first part of this chapter by showcasing the resulting workflow in the GitHub Web user interface (UI). In the second part of this section, we present the quantitative evaluation results of applying the prototype to the prepared repository containing the QuixBugs dataset.

6.1. Showcase of Workflow

Setting up the APR system in a repository is achieved by adding 2 files to the ".github" directory of the repository. The required files are ".github/workflows/auto-fix.yml" and ".github/scripts/filter_issues.py". Furthermore, an "LLM_API_KEY" secret needs to be added to the repository secrets, and GitHub Actions needs to be granted permission to create pull requests. With these in place, the system is ready to operate.
An optional configuration file can be added to the root of the repository. This '.bug-fix.yml" file can be used to override the LLM used, max attempts, and naming conventions for labels and branches.

With the system in place and a custom configuration file set up, the APR system is ready to be used in the repository. Bugs can be automatically fixed in two different ways.

One option is processing a single issue immediately when it is created and labeled with "bug_v01", as shown in Figure 6.1. This allows for fast feedback and quick bug fixing at issue creation and triage[10].

[10]Issue triage is the process of categorizing and prioritizing issues such as bug reports. [81]

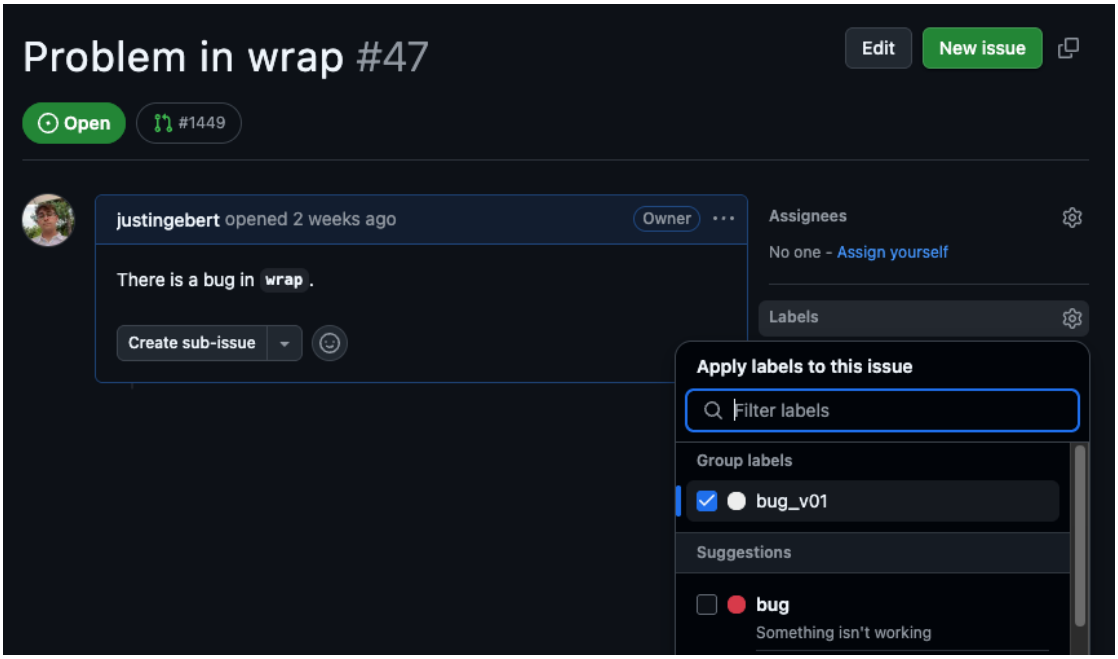

Figure 6.1.: *Trigger automatic fixing for single issue, Source: screenshot*

The second way fetches and processes all issues labeled with the "bug_v01" label. Scheduling the workflow at a specific time or dispatching it manually (see Figure 6.2) will result in such a run. This offers a more controlled approach to bug fixing, where issues are processed in batches.

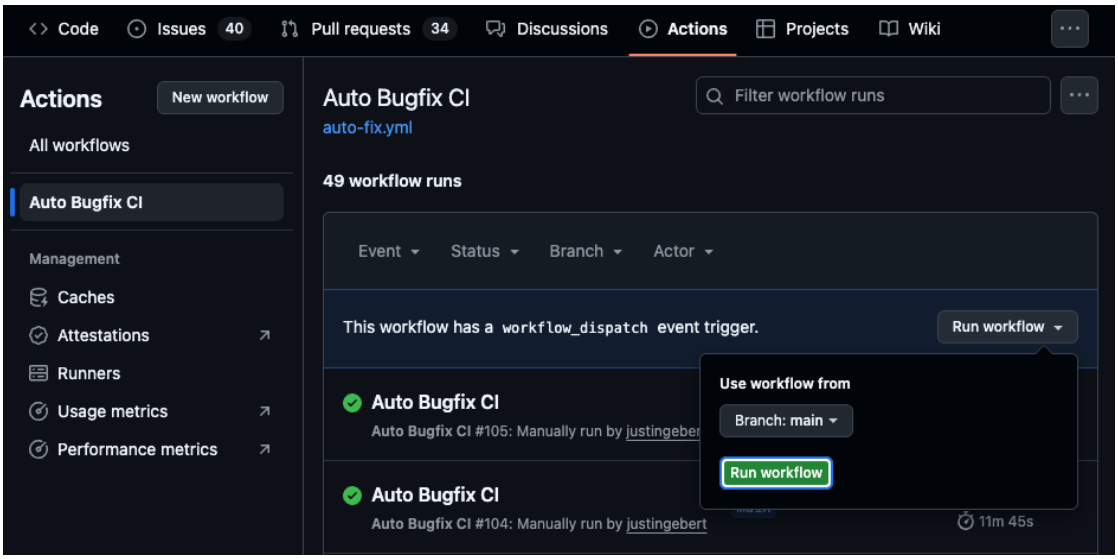

Figure 6.2.: *Manual Dispatch of APR, Source: screenshot*

When the workflow is triggered, it creates a new run in the GitHub Actions tab (see Figure 6.3). This executes the bug fixing logic described in Section 5.1.

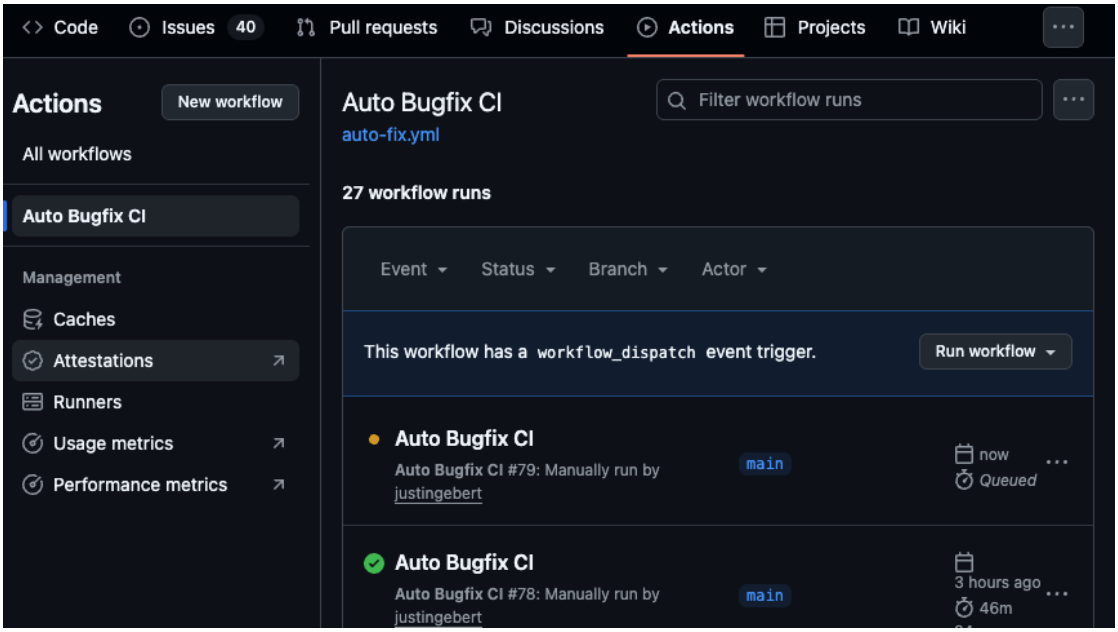

Figure 6.3.: *GitHub Action Run, Source: own representation*

A run of the APR system can result in two possible outcomes for every processed issue. If a repair attempt is successful, the system creates a pull request containing the proposed code changes and links it to the corresponding issue. This enables users to review the fix and merge it into the main branch. Upon merging, the issue is automatically closed. Figure 6.4 shows an example of a resulting pull request created by the APR system.

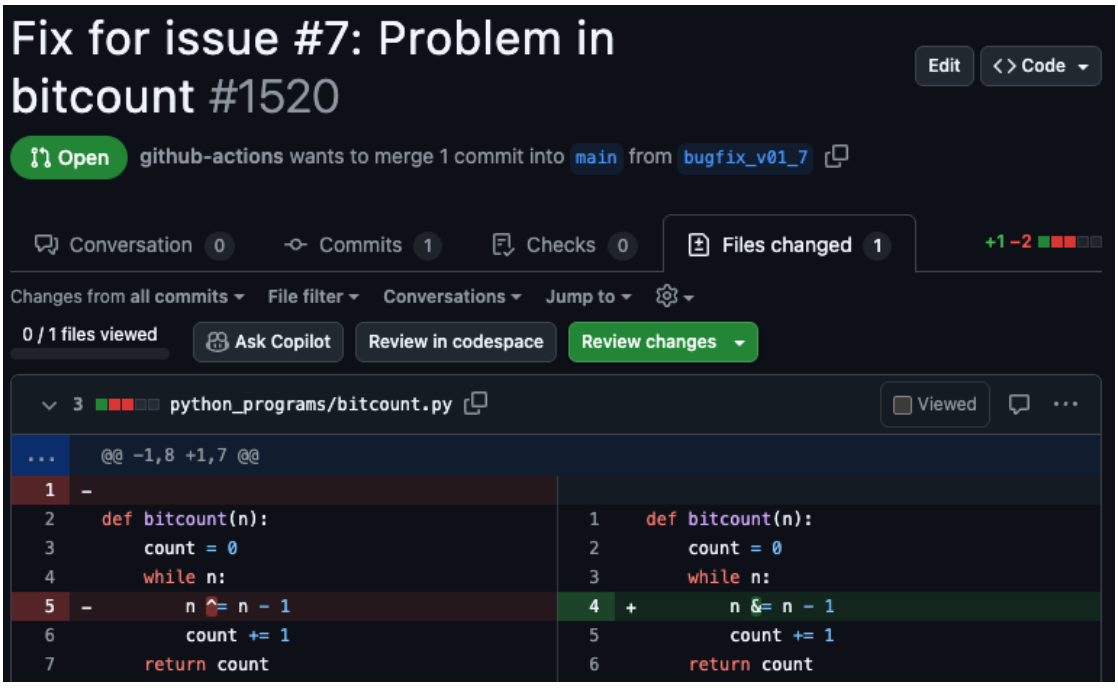

Figure 6.4.: *Resulting Pull Request, Source: screenshot*

When an issue repair fails after all attempts have been exhausted, the failure is reported to the issue as a comment and the issue is labeled as failed so it won't be picked up again (see Figure 6.5). This allows for easy tracking of issues that could not be fixed by the APR system.

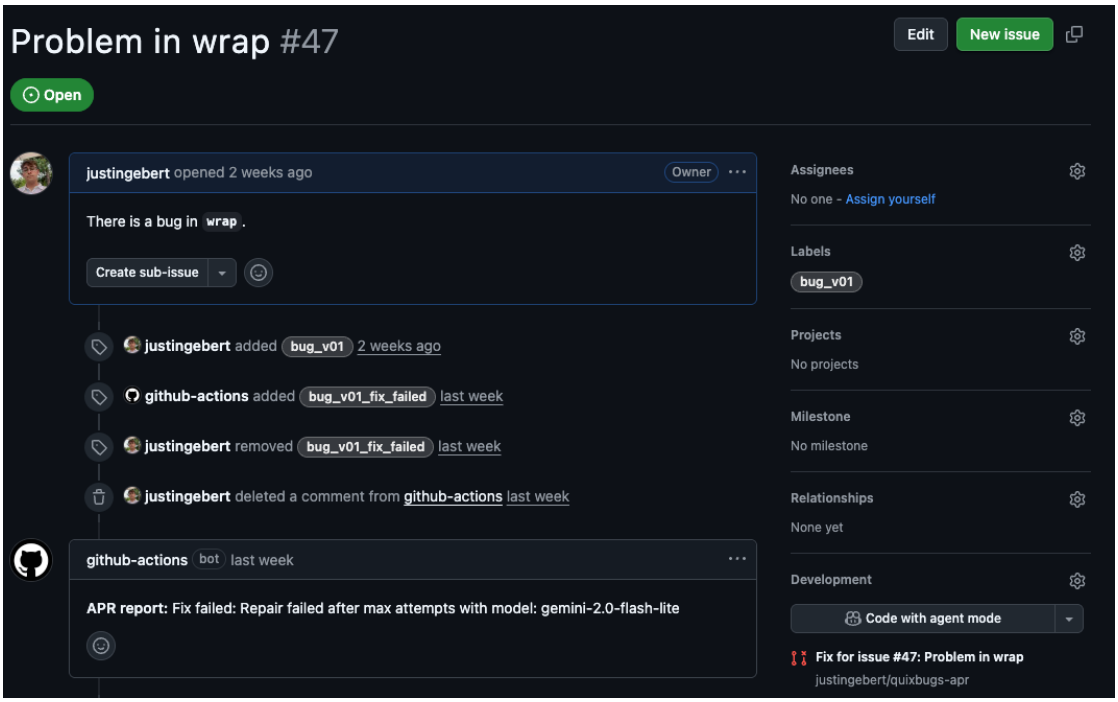

Figure 6.5.: *Failure Report, Source: screenshot*

Issues marked as failed can be picked up again by adding new context to the issue. This can be done by adding a comment to the issue, which will trigger the APR system to pick up the issue again and try to fix it with the added context. This allows for a more dynamic approach to bug fixing, where issues can be fixed as new information becomes available.

For transparency and debugging, each run provides a live log stream in the GitHub Actions tab (see Figure 6.6). This allows users to see the progress of the run and any errors that occur during the execution. For further analysis, logs, metrics, and the complete context objects are published as artifacts for each run, available to download in the run view.

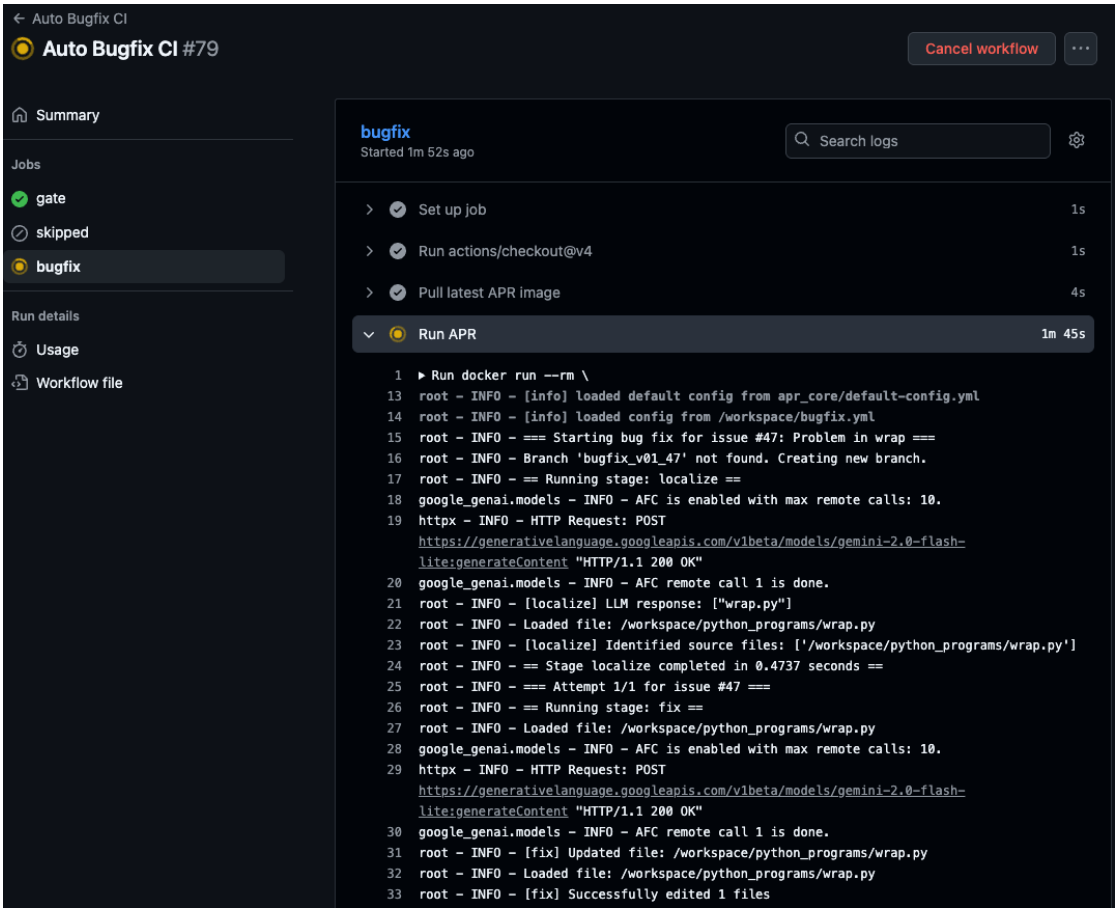

Figure 6.6.: *APR log stream, Source: screenshot*

Figure 6.7 visualizes the resulting workflow of the APR system. The diagram shows the relation between user actions (yellow) and results (blue).

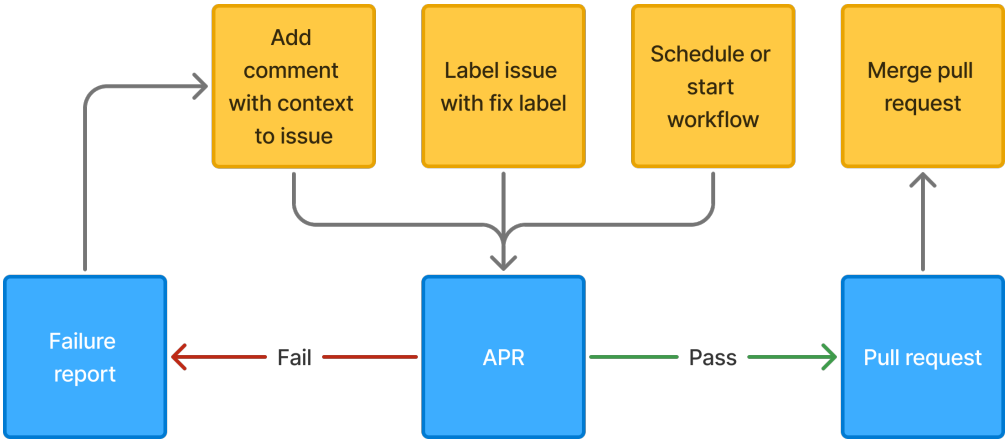

Figure 6.7.: *Resulting flow diagram, Source: own representation*

6.2. Evaluation Results

In this section, we present the results of the quantitative evaluation of the implemented APR prototype. The evaluation is based on the collected and calculated data for each run of the prototype. How this data was collected and calculated is described in Section 3.3. Threats to validity of these results are discussion in Section 7.1.

6.2.1. Baseline of Evaluation

The resulting data is based on 24 executions of the APR prototype in the prepared repository (see Section 3.1.3), which contains the QuixBugs benchmark. For evaluating the effectiveness, performance, and cost, we ran the APR prototype using twelve selected LLMs defined in Section 3.1.2. All models were tested with one attempt per issue to evaluate zero-shot performance and with a retry loop enabled to compare few-shot performance. This results in 24 pipeline runs each processing 40 issues from the repository.

6.2.2. Results

The following tables and diagrams show the repair success rate, average cost per issue, and average execution time per issue for each model used in the evaluation. Table 6.1 shows the results of the evaluation with one attempt per issue, while the Table 6.2 shows the results with the retry loop enabled and the max attempts set to 3. Diagrams 6.8, 6.9, and 6.10 visualize and compare the results of the evaluation for each model in a grouped manner. Diagrams 6.11 and 6.12 show the CI overhead for each model in relation to the execution time of the run.

Table 6.1.: *Zero shot evaluation results*

Model	ID	Repair Success Rate	Average Cost Per Issue	Average Execution Time Per Issue
gemini-2.0-flash-lite	G2F-L	82.5%	$0.0001	6.06s
gemini-2.0-flash	G2F	87.5%	$0.0002	8.18s
gemini-2.5-flash-lite	G25F-L	90.0%	$0.0002	5.07s
gemini-2.5-flash	G25F	92.5%	$0.009	20.46s
gemini-2.5-pro	G25P	95.0%	$0.0713	70.09s
gpt-4.1-nano	GPT4N	70.0%	$0.0001	6.73s
gpt-4.1-mini	GPT4M	90.0%	$0.0007	8.96s
gpt-4.1	GPT4	90.0%	$0.0033	7.42s
o4-mini	O4M	90.0%	$0.0069	23.08s
claude-3-5-haiku	C35H	0.0%	$0.0024	9.17s
claude-3-7-sonnet	C37S	87.5%	$0.0069	11.72s
claude-sonnet-4-0	CS40	90.0%	$0.0103	12.96s

Table 6.2.: *Few shot evaluation results*

Model	ID	Repair Success Rate	Average Cost Per Issue	Average Execution Time Per Issue
gemini-2.0-flash-lite	G2F-L	85%	$0.0003	13.08s
gemini-2.0-flash	G2F	90.0%	$0.0004	8.27s
gemini-2.5-flash-lite	G25F-L	90.0%	$0.0003	7.18s
gemini-2.5-flash	G25F	95.0%	$0.0111	23.65s
gemini-2.5-pro	G25P	97.5%	$0.0708	68.78s
gpt-4.1-nano	GPT4N	90.0%	$0.0003	10.61s
gpt-4.1-mini	GPT4M	97.5%	$0.0043	11.98s
gpt-4.1	GPT4	97.5%	$0.004	7.45s
o4-mini	O4M	100%	$0.007	21.54s
claude-3-5-haiku	C35H	15.0%	$0.0076	23.98s
claude-3-7-sonnet	C37S	95.0%	$0.0085	27.16s
claude-sonnet-4-0	CS40	92.5%	$0.0117	16.80s

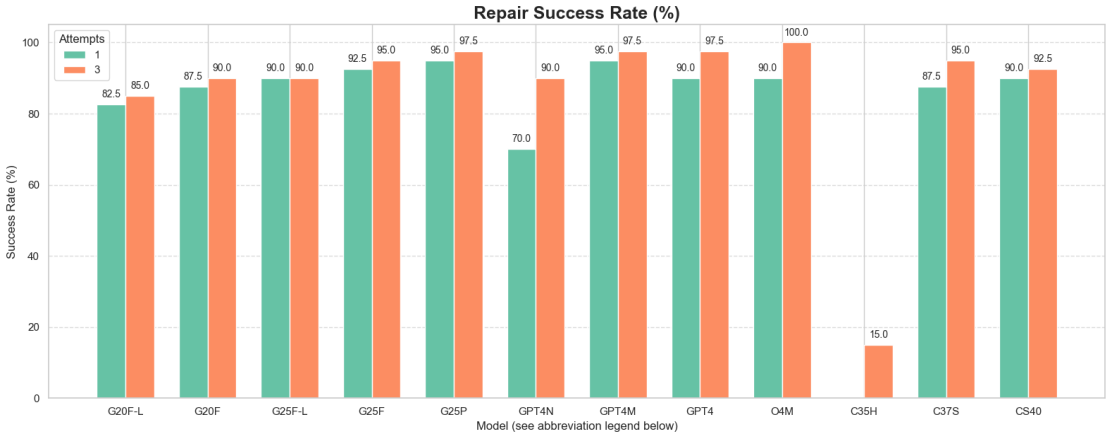

Figure 6.8.: *Repair Success Rate per Model, Source: own representation*

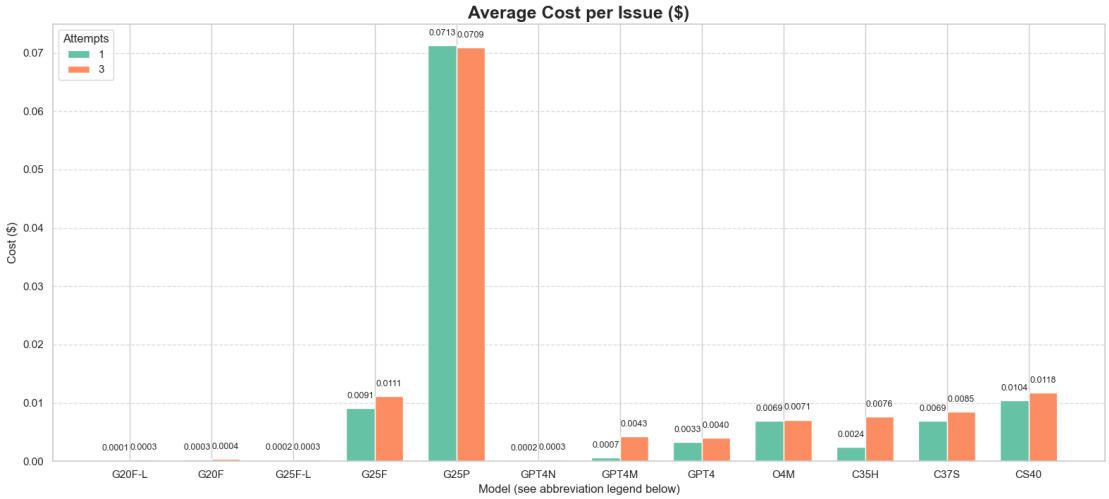

Figure 6.9.: *Average Cost per Issue per Model, Source: own representation*

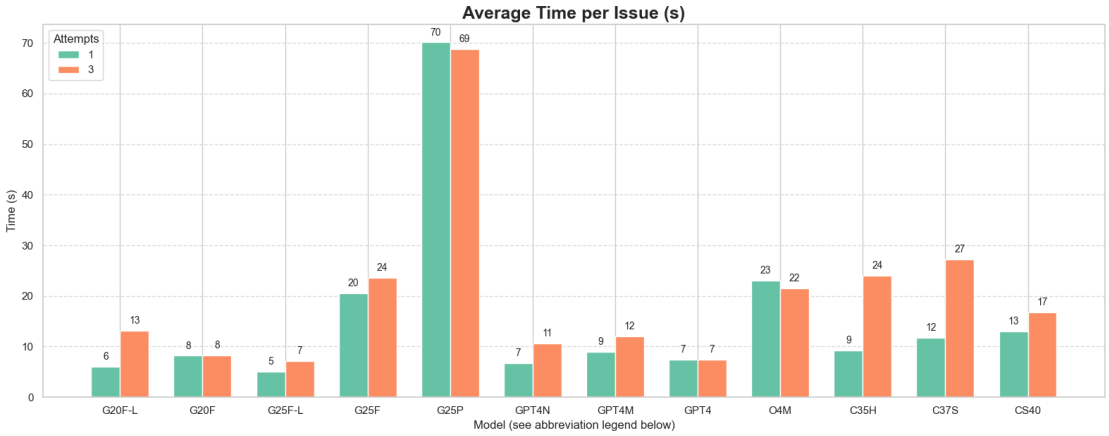

Figure 6.10.: *Average Execution Time per Issue per Model, Source: own representation*

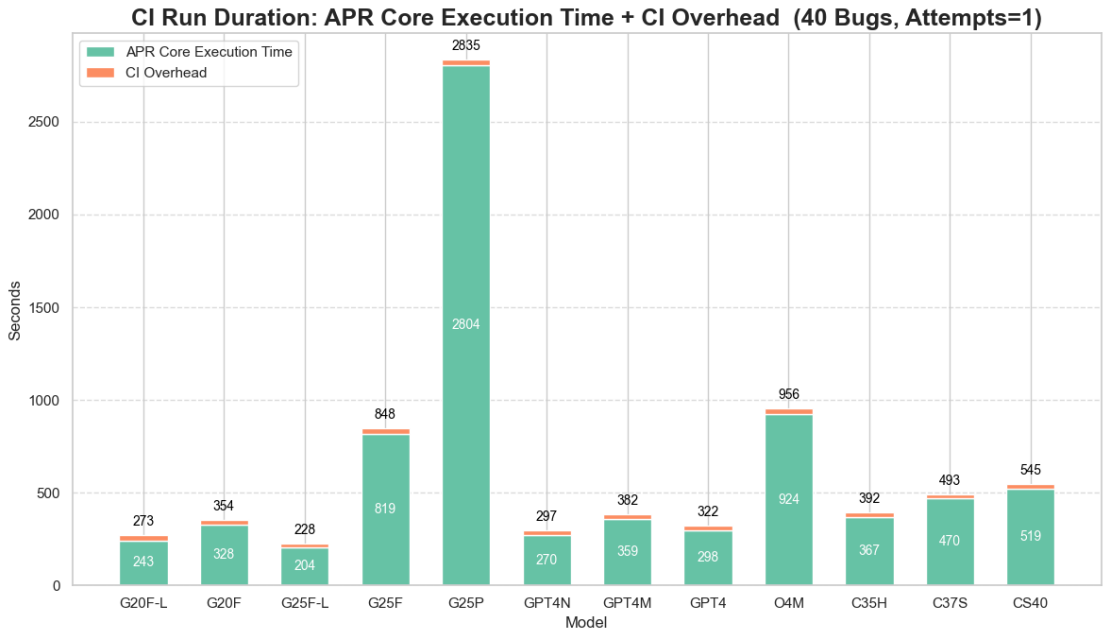

Figure 6.11.: *CI Overhead per Run with 1 Attempt, Source: own representation*

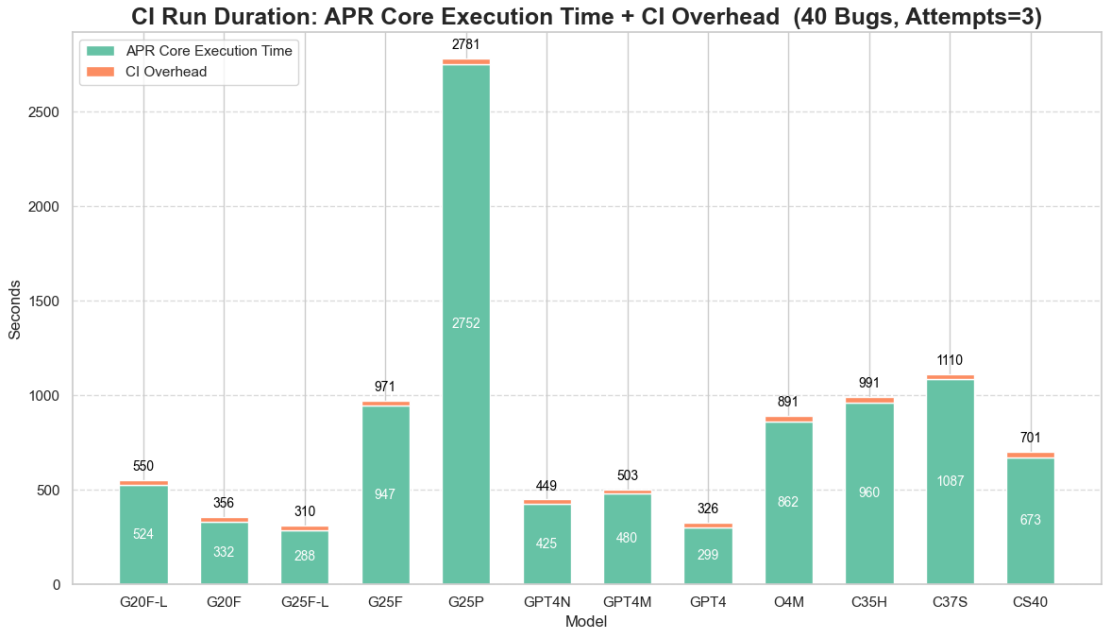

Figure 6.12.: *CI Overhead per Run with 3 Attempts, Source: own representation*

The full set of resulting data can be found in the "apr_evaluation" directory of the "quixbugs-apr" repository, listed in the Appendix A.

7. Discussion

In this section, we will discuss the results of the evaluation of our prototype and put them into context to answer the research questions. We begin by discussing the validity of our results, continue with the potentials and limitations of our approach and summarize the lessons learned. Finally, we outline a roadmap for future extensions of our work.

7.1. Validity

The evaluation results are very promising but face several limitations that affect their validity.

Since the repair process relies on LLMs, which are non-deterministic by design, executions may vary in their results. As the pipeline was only run twice per model, these results are not fully representative of each model's performance. Furthermore, the speed and availability of the tested LLMs heavily depends on the providers' APIs, which can cause fluctuating execution times or even failures during periods of high traffic.

Repair costs are calculated based on token usage reported by the providers' API responses. However, these numbers are not fully accurate, as providers can be non-transparent about actual token counts [82]. As a result, the reported costs and execution times should be interpreted with caution and are not reliably comparable across providers.

Additionally, the system was executed on GitHub-hosted runners included in the free tier of GitHub. Therefore, the performance metrics reflect the behavior of GitHub's cloud-hosted CI environment. While this allows for rapid prototyping and testing, it limits the generalizability of absolute execution times and costs.

The evaluation is based on the QuixBugs benchmark, which includes 40 single-line bugs, each in a separate file. This dataset does not fully represent real-world software development scenarios, as it only covers a narrow set of small algorithmic bugs. Moreover, we assume that passing the provided test suite indicates a correct fix. While QuixBugs offers relatively thorough tests for its scale, they do not guarantee semantic correctness or full behavioral equivalence with the ground truth. Therefore, a generated fix may pass all tests but still not fully resolve the underlying bug by introducing subtle errors that are not detected by the tests.

Another important consideration is that QuixBugs is an open-source benchmark published prior to the release of the evaluated LLMs. It is possible that the dataset or some of its bugs may have appeared in the training data of certain models. While this cannot be verified, it represents a further threat to validity and may inflate model performance.

To partially offset these limitations, we evaluated twelve diverse LLMs. These models vary in capability, speed and cost, which provides insight into how different configurations might generalize to larger datasets or other development environments. However, to draw broader conclusions about real-world effectiveness, further evaluation on more complex benchmarks such as Defects4J or SWE-Bench is necessary.

The threats outlined above limit the scope of our conclusions. Additional testing other programming languages, codebases, and CI platforms is required to fully validate the approach at production scale. Nevertheless, the results demonstrate that an LLM-based automated bug fixing pipeline can be successfully integrated into a CI workflow and can achieve non-trivial repair rates with minimal time investment at relatively low cost.

7.2. Potentials

Section 5 addresses RQ1 by demonstrating that LLM-based automated bug fixing can effectively be integrated into CI workflows through the use of GitHub Actions and containerized APR logic. While the prototype was used in two repositories[11], the underlying approach is designed for portability and can be extended to any Python project with minimal effort. The use of a YAML configuration file enables rapid adaptation of the system to new repositories and environments. This modularity allows for ongoing adjustments and improvements to the bug-fixing system without requiring changes to the codebase, making the solution both flexible and maintainable.

By showcasing the resulting workflow in Section 6.1, we demonstrated that, once integrated, producing an automated bug fix requires only a single user action. For example, a developer can simply create and label an issue[12], or schedule a dispatch within the GitHub repository. This highlights the potential of the approach to fully automate the bug-fixing process, eliminating the need for developer intervention in the repair workflow. Issue creation for tracking features, adjustments, or bugs is fundamental components of agile development. [21] By integrating with this workflow, our system allows automated bug fixes to be triggered directly from these standard agile practices.

The resulting workflow further indicates that this integration can actively support developers throughout the agile lifecycle by automating the design, development, and testing stages for bug fixes. Developers are left primarily with specifying requirements in the form of issues and reviewing the generated fixes, thus reducing their manual workload. Notably, the system is able to process and resolve issues even with minimal descriptions, as shown in our evaluation, which relied on issues with little detailed information. This allows developers to focus on more complex and creative tasks, while routine bug fixes are handled automatically. These findings are consistent with the work of Hou et al. [1], who states that LLMs can accelerate bug fixing and enhance both software reliability and maintainability.

[11] Evaluated in the "quixbugs-apr" and tested in "bugfix-ci" (see Appendix A)

[12] The issue must be assigned the according label (e.g., `bug_v01`) that the APR system is configured to process.

For answering RQ2, the evaluation results in Section 6.2 demonstrate that the prototype can achieve a repair success rate of up to 100% on the QuixBugs benchmark, with the best-performing model (o4-mini) successfully repairing all 40 bugs in the few-shot configuration. This highlights the effectiveness of the proposed approach for fixing single-file bugs within a CI environment. However, a deeper look into the collected data reveals that the repair success rate varies across different LLMs and configurations, indicating a strong dependence on the LLM used for repair.

In the zero-shot configuration, where each issue receives only a single repair attempt, several models already achieve high repair success rates. For example, gemini-2.5-pro achieves a success rate of 95% (38 out of 40 issues), while gpt-4.1-mini and claude-sonnet-4-0 reach 90%. Lightweight models such as gemini-2.0-flash-lite and gemini-2.5-flash-lite achieve success rates of 82.5% and 90%, respectively. These results indicate that smaller models can still deliver strong performance in zero-shot settings, although the highest success rates are achieved by more capable models.

The main advantage of smaller models lies in their cost-effectiveness and rapid execution. For instance, gemini-2.5-flash-lite achieves a repair success rate of 90% in the zero-shot setting, with an average cost per issue of just $0.0002 and an average execution time of 5.07 seconds. Similarly, gemini-2.0-flash-lite repairs 82.5% of issues at an extremely low cost of $0.0001 per issue and an average time of 6.07 seconds. In comparison, larger models such as gemini-2.5-pro reach a 95% repair rate but at higher costs ($0.071 per issue) and longer execution times (70.09 seconds per issue). These findings indicate that smaller more cost-efficient models can be highly effective for automated bug fixing in CI environments, enabling frequent and rapid submissions of fixes at minimal cost.

Allowing a LLM to make multiple repair attempts per issue (few-shot), using internal feedback loops, significantly improves repair success rates across all models. For example, with up to three attempts, the o4-mini model improves from 90% to 100% repair success, and gpt-4.1-mini increases from 95% to 97.5%. Smaller models such as gpt-4.1-nano show an even larger improvement, rising from 70% in the single-attempt setting to 90% with retries. Importantly, these increases in success rate are achieved with a rise in cost and execution time. As a result, even lightweight models can reach performance levels similar to larger models while maintaining the advantages of lower cost and faster execution. These findings of improved few-shot performance align with the observations of Brown et al. [43], who demonstrated the effectiveness of few-shot learning for large language models.

The measured repair times across all models range from as low as 5 seconds per issue (e.g., gemini-2.5-flash-lite) up to about 70 seconds per issue (gemini-2.5-pro). When looking at the total execution time for the entire pipeline, the results show that CI overhead is minimal. This indicates that the integration of LLM-based bug fixing into CI workflows does not introduce significant bottlenecks or computational overhead. All pipeline runs complete single issue repairs well within typical CI job expectations[13], meaning the approach does not impose significant waiting times for developers.

[13]Recent research suggests that 10 minutes is the most acceptable build duration [83].

In terms of cost, repair attempts vary from as little as $0.0001 per issue for lightweight models to around $0.07 per issue for larger models. These results confirm that automated bug fixing with LLMs is both time-efficient and cost-effective, making it practical for integration into CI environments without introducing workflow bottlenecks or major expenses.

Overall, the combination of high repair success rates, low execution times, and minimal costs demonstrates that this approach is both feasible and practical for automating bug fixing in CI environments. Because each repair task is isolated, multiple issues can be processed concurrently, further increasing throughput and minimizing delays for development.

By implementing a lightweight approach together with the interactivity of the GitHub platform, this system achieves repair success rates on the QuixBugs benchmark that are comparable to those reported in recent APR research [16, 54]. In addition, the direct integration into the CI workflow provides transparent metrics for both cost and performance, offering practical advantages over many traditional APR approaches.

As highlighted in the results, system effectiveness and efficiency are highly dependent on the underlying LLM used. However, the modular design of the approach allows for straightforward integration of improved or newly released models. This ensures that the system can benefit from advances in LLM technology without requiring major changes to the overall pipeline. Its portability and modularity allow for a wide range of enhancements, such as integrating new LLMs, experimenting with different prompting strategies or incorporating richer feedback mechanisms, as described in in Section 7.5.

7.3. Limitations

Ultimately, there are also limitations faced by the implementation and the overall approach.

As mentioned in section 7.1, the system is constrained to addressing small issues only. Even with small issues, the performance and availability of the system highly depend on external factors such as the LLM providers' APIs and the GitHub Actions runners. This presents a limitation in terms of reliability and performance in a real software development cycle.

Regarding integration with GitHub, the system faces additional limitations imposed by the GitHub Actions environment. Workflow runs cannot be skipped, and the filtering logic for events using external configuration data is limited. As a result, the workflow must execute on every issue labeling event to perform filtering in the first job. This causes the GitHub Actions tab to fill with runs that are marked as successful but did not process any bugs, resulting in a cluttered run history and potential confusion for users. This could be improved by migrating the APR Core to a GitHub App integration that listens to webhook events and only triggers the workflow for relevant issues.

Additionally, security and privacy concerns arise from the fact that the program repair relies on LLMs. Since the issue title and description are added to the prompt used

for repair, malicious instructions could be inserted into an issue. Therefore, untrusted project contributors should not be allowed to create or edit issues. Furthermore, code submitted as a pull request must be carefully verified and reviewed before merging.

Lastly, LLM providers like Google, OpenAI, and Anthropic are not fully transparent about their data and storage policies. This may raise concerns about the privacy of code and issues processed by the system, especially for private or sensitive repositories. While the prototype currently relies on commercial models, its modular design allows for straightforward integration of open source alternatives in the future. Nevertheless, copyright and licensing issues may arise when using code generated by LLMs trained on copyrighted data [49, 1].

7.4. Lessons Learned

The development and evaluation of this prototype provided several valuable insights, both technical and practical:

- **Effectiveness of LLMs:** Large Language Models can automate bug fixing in CI environments, but outcomes are highly dependent on the model's capabilities, context and prompting techniques.
- **Provider and API Reliability:** The APIs of LLM providers are not fully reliable, with some, such as Googles Gemini-2.5-pro, experiencing instabilities or high latency. Developing retry logic and proper error handling is essential for using external APIs.
- **Rapid Evolution:** The landscape of AI is evolving quickly, with new models and approaches released on a daily basis. Following research and designing modular software are important to stay up to date in this field.
- **Benchmarking Challenges:** Benchmark datasets like QuixBugs provide a starting point for evaluation, but real-world applications will require broader benchmarks like SWE Bench with diverse scenarios to fully assess effectiveness.
- **Human Oversight Remains Important:** While automation can handle routine bugs, human review remains important, particularly for accessing fixes for edge cases, hallucinations and insecurities.

Overall, the implementation revealed both the potential and the current limitations of integrating LLM-based program repair into CI and provided valuable insights in the field of APR.

7.5. Roadmap for Extensions

The modular and extensible design of this prototype allows for future improvements, both in research and practical application. The ideas presented in this section have emerged during implementation and while exploring the latest developments in APR research discussed throughout this thesis. Due to time constraints, these directions were not realized, but they represent valuable opportunities for further work.

- **Deeper Data Analysis:** The system already collects logs and metrics, which could be analyzed in greater detail to identify failures and improve localization and repair.
- **Optimization of Cost, Time, and Success Rates:** More extensive experimentation with models, context engineering and prompting strategies could help to further optimize costs and execution times while maximizing repair success rates.
- **Adaptive Model Selection:** Implementing a dynamic approach where lightweight models handle most repairs but challenging cases fall back to more powerful models could improve effectiveness and efficiency.
- **Broader and Richer Benchmarks:** Extending evaluation beyond QuixBugs to include larger and more diverse benchmarks, such as Defects4J or SWE-bench, would provide a stronger basis for generalization and reveal strengths and limitations in real-world settings.
- **Improved Platform Integration:** Integrating the system as a GitHub App that leverages webhook events, or using service accounts for better access control, could simplify development and improve user experience.
- **Experimentation with Agent Architectures:** Testing and comparing multi-agent or interactive agent approaches could uncover additional gains in repair effectiveness.
- **Human Factors and Developer Experience:** Future work could measure developer trust, satisfaction, and acceptance of automated fixes in real teams, supporting adoption in production workflows.

These directions illustrate just a few of the many ways the presented approach can serve as a foundation for continued innovation in automated program repair (APR).

8. Conclusion

In conclusion, this thesis explored the integration of LLM-based automated bug fixing within continuous integration (CI) with a focus on implementation and evaluation of potentials and limitations of the approach. The developed prototype successfully demonstrated a seamless integration of an automated bug fixing solution into a GitHub repository, using GitHub Actions and containerization. This integration automates the process from issue creation to pull request generation, showcasing the potential of LLMs in modern software development workflows.

The evaluation on the QuixBugs benchmark showed promising results, with repair success rates of up to 100% by using iterative prompting techniques leveraging execution environment information. Testing twelve different popular LLMs revealed that smaller, more cost-effective models can achieve good performance for small bugs. These models balance high repair success rates with rapid execution times and minimal costs, thus proving suitable for frequent integration into routine software development practices.

However, we also identified challenges and limitations of the integration, including the reliance on external LLM provider APIs, potential security and privacy vulnerabilities and restricted generalizability given the limited scope of the QuixBugs benchmark. Additionally, technical constraints of the GitHub Action CI platform can lead to workflow management issues, such as cluttered run histories.

Ultimately, this thesis contributes a practical, effective, and scalable framework for incorporating automated bug fixing into real-world software development environments, laying a foundation for future innovations and broader adoption of AI-driven APR methodologies.

References

[1] Hou, Xinyi et al. *Large Language Models for Software Engineering: A Systematic Literature Review*. Apr. 2024. DOI: 10.48550/arXiv.2308.10620. arXiv: 2308.10620 [cs]. (Visited on 03/06/2025).

[2] Puvvadi, Meghana et al. "Coding Agents: A Comprehensive Survey of Automated Bug Fixing Systems and Benchmarks". In: *2025 IEEE 14th International Conference on Communication Systems and Network Technologies (CSNT)*. Mar. 2025, pp. 680–686. DOI: 10.1109/CSNT64827.2025.10968728. (Visited on 04/27/2025).

[3] Winter, Emily et al. "How Do Developers Really Feel About Bug Fixing? Directions for Automatic Program Repair". In: *IEEE Transactions on Software Engineering* 49.4 (Apr. 2023), pp. 1823–1841. ISSN: 1939-3520. DOI: 10.1109/TSE.2022.3194188. (Visited on 06/24/2025).

[4] Vasilescu, Bogdan et al. "The Sky Is Not the Limit: Multitasking across GitHub Projects". In: *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas: ACM, May 2016, pp. 994–1005. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884875. (Visited on 06/24/2025).

[5] Tihanyi, Norbert et al. *A New Era in Software Security: Towards Self-Healing Software via Large Language Models and Formal Verification*. June 2024. DOI: 10.48550/arXiv.2305.14752. arXiv: 2305.14752 [cs]. (Visited on 06/26/2025).

[6] *Cost of Poor Software Quality in the U.S.: A 2022 Report*. (Visited on 06/24/2025).

[7] Zhang, Feng et al. "An Empirical Study on Factors Impacting Bug Fixing Time". In: *2012 19th Working Conference on Reverse Engineering*. Oct. 2012, pp. 225–234. DOI: 10.1109/WCRE.2012.32. (Visited on 06/24/2025).

[8] Lee, Cheryl et al. *A Unified Debugging Approach via LLM-Based Multi-Agent Synergy*. Oct. 2024. DOI: 10.48550/arXiv.2404.17153. arXiv: 2404.17153 [cs]. (Visited on 03/06/2025).

[9] Xia, Chunqiu Steven et al. *Agentless: Demystifying LLM-based Software Engineering Agents*. Oct. 2024. DOI: 10.48550/arXiv.2407.01489. arXiv: 2407.01489 [cs]. (Visited on 04/24/2025).

[10] Zhang, Yuwei et al. "PATCH: Empowering Large Language Model with Programmer-Intent Guidance and Collaborative-Behavior Simulation for Automatic Bug Fixing". In: *ACM Transactions on Software Engineering and Methodology* (Feb. 2025), p. 3718739. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3718739. (Visited on 03/24/2025).

[11] Wang, Jialin and Duan, Zhihua. *Empirical Research on Utilizing LLM-based Agents for Automated Bug Fixing via LangGraph*. Jan. 2025. DOI: 10.33774/coe-2025-jbpg6. (Visited on 03/12/2025).

[12] Liu, Yizhou et al. *MarsCode Agent: AI-native Automated Bug Fixing*. Sept. 2024. DOI: 10.48550/arXiv.2409.00899. arXiv: 2409.00899 [cs]. (Visited on 03/06/2025).

[13] Yang, John et al. *SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering*. Nov. 2024. DOI: 10.48550/arXiv.2405.15793. arXiv: 2405.15793 [cs]. (Visited on 04/20/2025).

[14] Sobania, Dominik et al. "An Analysis of the Automatic Bug Fixing Performance of ChatGPT". In: *2023 IEEE/ACM International Workshop on Automated Program Repair (APR)*. May 2023, pp. 23–30. DOI: 10.1109/APR59189.2023.00012. (Visited on 03/06/2025).

[15] Xia, Chunqiu Steven and Zhang, Lingming. "Automated Program Repair via Conversation: Fixing 162 out of 337 Bugs for $0.42 Each Using ChatGPT". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Vienna Austria: ACM, Sept. 2024, pp. 819–831. ISBN: 979-8-4007-0612-7. DOI: 10.1145/3650212.3680323. (Visited on 05/12/2025).

[16] Hu, Haichuan et al. *Can GPT-O1 Kill All Bugs? An Evaluation of GPT-Family LLMs on QuixBugs*. Dec. 2024. DOI: 10.48550/arXiv.2409.10033. arXiv: 2409.10033 [cs]. (Visited on 04/15/2025).

[17] Rondon, Pat et al. *Evaluating Agent-based Program Repair at Google*. Jan. 2025. DOI: 10.48550/arXiv.2501.07531. arXiv: 2501.07531 [cs]. (Visited on 03/24/2025).

[18] Chen, Zhi, Ma, Wei, and Jiang, Lingxiao. *Unveiling Pitfalls: Understanding Why AI-driven Code Agents Fail at GitHub Issue Resolution*. Mar. 2025. DOI: 10.48550/arXiv.2503.12374. arXiv: 2503.12374 [cs]. (Visited on 03/24/2025).

[19] Ugwueze, Vincent and Chukwunweike, Joseph. "Continuous Integration and Deployment Strategies for Streamlined DevOps in Software Engineering and Application Delivery". In: *International Journal of Computer Applications Technology and Research* (Jan. 2024), pp. 1–24. DOI: 10.7753/IJCATR1401.1001.

[20] Ruparelia, Nayan B. "Software Development Lifecycle Models". In: *ACM SIGSOFT Software Engineering Notes* 35.3 (May 2010), pp. 8–13. ISSN: 0163-5948. DOI: 10.1145/1764810.1764814. (Visited on 06/25/2025).

[21] Abrahamsson, Pekka et al. *Agile Software Development Methods: Review and Analysis*. Sept. 2017. DOI: 10.48550/arXiv.1709.08439. arXiv: 1709.08439 [cs]. (Visited on 06/25/2025).

[22] Zayat, Wael and Senvar, Ozlem. "Framework Study for Agile Software Development Via Scrum and Kanban". In: *International Journal of Innovation and Technology Management* 17.04 (June 2020). ISSN: 0219-8770, 1793-6950. DOI: 10.1142/s0219877020300025. (Visited on 07/20/2025).

[23] Huo, Ming et al. "Software Quality and Agile Methods". In: *Proceedings of the 28th Annual International Computer Software and Applications Conference, 2004. COMPSAC 2004.* Sept. 2004, 520–525 vol.1. DOI: 10.1109/CMPSAC.2004.1342889. (Visited on 07/20/2025).

[24] *7 Mistakes I've Learned during 7 Years of Software Development*. URL: https://www.lunadio.com/blog/7-mistakes-ive-learned-during-7-years-of-software-development/. Aug. 2020. (Visited on 07/29/2025).

[25] Francesco, Paolo Di, Malavolta, Ivano, and Lago, Patricia. "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption". In: *2017 IEEE International Conference on Software Architecture (ICSA)*. Apr. 2017, pp. 21–30. DOI: 10.1109/ICSA.2017.24. (Visited on 07/20/2025).

[26] Tregubov, Alexey et al. "Impact of Task Switching and Work Interruptions on Software Development Processes". In: *Proceedings of the 2017 International Conference on Software and System Process*. Paris France: ACM, July 2017, pp. 134–138. ISBN: 978-1-4503-5270-3. DOI: 10.1145/3084100.3084116. (Visited on 06/23/2025).

[27] Lovett, Channing. *Top 12 Benefits of Continuous Integration*. Aug. 2023. (Visited on 07/29/2025).

[28] Hilton, Michael et al. "Usage, Costs, and Benefits of Continuous Integration in Open-Source Projects". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. Singapore Singapore: ACM, Aug. 2016, pp. 426–437. DOI: 10.1145/2970276.2970358. (Visited on 07/20/2025).

[29] Ghaleb, Taher Ahmed, Da Costa, Daniel Alencar, and Zou, Ying. "An Empirical Study of the Long Duration of Continuous Integration Builds". In: *Empirical Software Engineering* 24.4 (Aug. 2019), pp. 2102–2139. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-019-09695-9. (Visited on 07/20/2025).

[30] Staff, GitHub. *Octoverse: AI Leads Python to Top Language as the Number of Global Developers Surges*. Oct. 2024. (Visited on 07/20/2025).

[31] *GitHub Features*. URL: https://github.com/features. 2025. (Visited on 07/20/2025).

[32] *About Issues*. URL: https://docs-internal.github.com/_next/data/LM3KAkw0Ii4E7Nz5N0zHV/en/free-pro-team%40latest/issues/tracking-your-work-with-issues/about-issues.json?versionId=free-pro-team%40latest&productId=issues&restPage=tracking-your-work-with-issues&restPage=about-issues. (Visited on 07/20/2025).

[33] *What Is Diff - Understanding Code Diff Tools*. URL: https://www.codecademy.com/article/what-is-diffing. (Visited on 07/25/2025).

[34] *About Pull Requests*. URL: https://docs-internal.github.com/_next/data/LM3KAkw0Ii4E7Nz5N0zHV/en/free-pro-team%40latest/pull-requests/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests.json?versionId=free-pro-team%40latest&productId=pull-requests&restPage=collaborating-with-pull-requests&restPage=proposing-changes-to-your-work-with-pull-requests&restPage=about-pull-requests. (Visited on 07/20/2025).

[35] *The Official YAML Web Site*. URL: https://yaml.org/. (Visited on 07/25/2025).

[36] *Understanding GitHub Actions*. URL: https://docs-internal.github.com/en/actions/get-started/understand-github-actions. (Visited on 07/29/2025).

[37] *GitHub Actions*. URL: https://github.com/features/actions. 2025. (Visited on 07/20/2025).

[38] *What Is Generative AI?* URL: https://research.ibm.com/blog/what-is-generative-AI. Feb. 2021. (Visited on 07/20/2025).

[39] Chang, Yupeng et al. "A Survey on Evaluation of Large Language Models". In: *ACM Transactions on Intelligent Systems and Technology* 15.3 (June 2024), pp. 1–45. ISSN: 2157-6904, 2157-6912. DOI: 10.1145/3641289. (Visited on 07/02/2025).

[40] Naveed, Humza et al. *A Comprehensive Overview of Large Language Models*. Oct. 2024. DOI: 10.48550/arXiv.2307.06435. arXiv: 2307.06435 [cs]. (Visited on 07/02/2025).

[41] Kaplan, Jared et al. *Scaling Laws for Neural Language Models*. Jan. 2020. DOI: 10.48550/arXiv.2001.08361. arXiv: 2001.08361 [cs]. (Visited on 07/21/2025).

[42] *LLMs: What's a Large Language Model? | Machine Learning | Google for Developers*. URL: https://developers.google.com/machine-learning/crash-course/llm/transformers. (Visited on 07/03/2025).

[43] Brown, Tom et al. "Language Models Are Few-Shot Learners". In: *Advances in Neural Information Processing Systems*. Vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. (Visited on 07/21/2025).

[44] *Introducing ChatGPT*. URL: https://openai.com/index/chatgpt/. Mar. 2024. (Visited on 07/25/2025).

[45] Dohmke, Thomas. *GitHub Copilot: Meet the New Coding Agent*. May 2025. (Visited on 06/26/2025).

[46] Bhargav Mallampati. "The Role of Generative AI in Software Development: Will It Replace Developers?" In: *World Journal of Advanced Research and Reviews* 26.1 (Apr. 2025), pp. 2972–2977. ISSN: 25819615. DOI: 10.30574/wjarr.2025.26.1.1387. (Visited on 06/25/2025).

[47] Kalliamvakou, Eirini. *Research: Quantifying GitHub Copilot's Impact on Developer Productivity and Happiness*. Sept. 2022. (Visited on 06/26/2025).

[48] Liu, Yi et al. *Prompt Injection Attack against LLM-integrated Applications*. Mar. 2024. DOI: 10.48550/arXiv.2306.05499. arXiv: 2306.05499 [cs]. (Visited on 07/03/2025).

[49] Sauvola, Jaakko et al. "Future of Software Development with Generative AI". In: *Automated Software Engineering* 31.1 (May 2024), p. 26. ISSN: 0928-8910, 1573-7535. DOI: 10.1007/s10515-024-00426-z. (Visited on 06/25/2025).

[50] *Introducing Codex*. URL: https://openai.com/index/introducing-codex/. (Visited on 06/26/2025).

[51] Zhang, Quanjun et al. "A Survey of Learning-based Automated Program Repair". In: *ACM Transactions on Software Engineering and Methodology* 33.2 (Feb. 2024), pp. 1–69. ISSN: 1049-331X, 1557-7392. DOI: 10.1145/3631974. (Visited on 06/26/2025).

[52] Bader, Johannes et al. "Getafix: Learning to Fix Bugs Automatically". In: *Proceedings of the ACM on Programming Languages* 3.OOPSLA (Oct. 2019), pp. 1–27. ISSN: 2475-1421. DOI: 10.1145/3360585. (Visited on 03/06/2025).

[53] Xia, Chunqiu Steven, Wei, Yuxiang, and Zhang, Lingming. "Automated Program Repair in the Era of Large Pre-trained Language Models". In: *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. Melbourne, Australia: IEEE, May 2023, pp. 1482–1494. ISBN: 978-1-6654-5701-9. DOI: 10.1109/ICSE48619.2023.00129. (Visited on 06/19/2025).

[54] Yin, Xin et al. "ThinkRepair: Self-Directed Automated Program Repair". In: *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. Vienna Austria: ACM, Sept. 2024, pp. 1274–1286. ISBN: 979-8-4007-0612-7. DOI: 10.1145/3650212.3680359. (Visited on 03/12/2025).

[55] Le Goues, Claire et al. "GenProg: A Generic Method for Automatic Software Repair". In: *IEEE Transactions on Software Engineering* 38.1 (Jan. 2012), pp. 54–72. ISSN: 1939-3520. DOI: 10.1109/TSE.2011.104. (Visited on 07/03/2025).

[56] Mechtaev, Sergey, Yi, Jooyong, and Roychoudhury, Abhik. "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis". In: *Proceedings of the 38th International Conference on Software Engineering*. Austin Texas: ACM, May 2016, pp. 691–701. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884807. (Visited on 07/03/2025).

[57] Tang, Yiting. "Large Language Models Meet Automated Program Repair: Innovations, Challenges and Solutions". In: *Applied and Computational Engineering* 117.1 (Dec. 2024), pp. 22–30. ISSN: 2755-2721, 2755-273X. DOI: 10.54254/2755-2721/2024.18303. (Visited on 06/01/2025).

[58] Lutellier, Thibaud et al. "CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair". In: *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. Virtual Event USA: ACM, July 2020, pp. 101–114. ISBN: 978-1-4503-8008-9. DOI: 10.1145/3395363.3397369. (Visited on 07/04/2025).

[59] Zhu, Qihao et al. "A Syntax-Guided Edit Decoder for Neural Program Repair". In: *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Athens Greece: ACM, Aug. 2021, pp. 341–353. ISBN: 978-1-4503-8562-6. DOI: 10.1145/3468264.3468544. (Visited on 07/04/2025).

[60] Xia, Chunqiu Steven and Zhang, Lingming. "Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning". In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Singapore Singapore: ACM, Nov. 2022, pp. 959–971. ISBN: 978-1-4503-9413-0. DOI: 10.1145/3540250.3549101. (Visited on 07/04/2025).

[61] Hossain, Soneya Binta et al. "A Deep Dive into Large Language Models for Automated Bug Localization and Repair". In: *Proceedings of the ACM on Software Engineering* 1.FSE (July 2024), pp. 1471–1493. ISSN: 2994-970X. DOI: 10.1145/3660773. (Visited on 03/13/2025).

[62] Anand, Avinash et al. *A Comprehensive Survey of AI-Driven Advancements and Techniques in Automated Program Repair and Code Generation*. Nov. 2024. DOI: 10.48550/arXiv.2411.07586. arXiv: 2411.07586 [cs]. (Visited on 06/01/2025).

[63] Meng, Xiangxin et al. *An Empirical Study on LLM-based Agents for Automated Bug Fixing*. Nov. 2024. DOI: 10.48550/arXiv.2411.10213. arXiv: 2411.10213 [cs]. (Visited on 03/06/2025).

[64] Meem, Fairuz Nawer, Smith, Justin, and Johnson, Brittany. "Exploring Experiences with Automated Program Repair in Practice". In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. Lisbon Portugal: ACM, Apr. 2024, pp. 1–11. ISBN: 979-8-4007-0217-4. DOI: 10.1145/3597503.3639182. (Visited on 06/26/2025).

[65] Wang, Kaixin et al. *Software Development Life Cycle Perspective: A Survey of Benchmarks for Code Large Language Models and Agents*. May 2025. DOI: 10.48550/arXiv.2505.05283. arXiv: 2505.05283 [cs]. (Visited on 07/04/2025).

[66] Lin, Derrick et al. "QuixBugs: A Multi-Lingual Program Repair Benchmark Set Based on the Quixey Challenge". In: *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. Vancouver BC Canada: ACM, Oct. 2017, pp. 55–56. ISBN: 978-1-4503-5514-8. DOI: 10.1145/3135932.3135941. (Visited on 04/17/2025).

[67] Just, René, Jalali, Darioush, and Ernst, Michael D. "Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs". In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. San Jose CA USA: ACM, July 2014, pp. 437–440. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2628055. (Visited on 07/04/2025).

[68] Le Goues, Claire et al. "The ManyBugs and IntroClass Benchmarks for Automated Repair of C Programs". In: *IEEE Transactions on Software Engineering* 41.12 (Dec. 2015), pp. 1236–1256. ISSN: 1939-3520. DOI: 10.1109/TSE.2015.2454513. (Visited on 07/04/2025).

[69] Jimenez, Carlos E. et al. *SWE-bench: Can Language Models Resolve Real-World GitHub Issues?* Nov. 2024. DOI: 10.48550/arXiv.2310.06770. arXiv: 2310.06770 [cs]. (Visited on 03/06/2025).

[70] *SWE-bench Lite*. URL: https://www.swebench.com/lite.html. (Visited on 07/30/2025).

[71] *Gemini Models | Gemini API*. URL: https://ai.google.dev/gemini-api/docs/models. (Visited on 07/24/2025).

[72] *Models - OpenAI API*. URL: https://platform.openai.com. (Visited on 07/24/2025).

[73] *Models Overview*. URL: https://docs.anthropic.com/en/docs/about-claude/models/overview. (Visited on 07/24/2025).

[74] Cohn, Mike. *User Stories Applied: For Agile Software Development*. USA: Addison Wesley Longman Publishing Co., Inc., 2004. ISBN: 0-321-20568-5.

[75] *GitHub's Plans*. URL: https://docs-internal.github.com/en/get-started/learning-about-github/githubs-plans. (Visited on 07/30/2025).

[76] *What Is an Image?* URL: https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-an-image/. 8:50:11 -0800 -0800. (Visited on 07/28/2025).

[77] *Black 25.1.0 Documentation*. URL: https://black.readthedocs.io/en/stable/#the-uncompromising-code-formatter. (Visited on 07/29/2025).

[78] *Flake8: Your Tool For Style Guide Enforcement — Flake8 7.3.0 Documentation*. URL: https://flake8.pycqa.org/en/latest/#. (Visited on 07/29/2025).

[79] *Workflow Syntax for GitHub Actions*. URL: https://docs-internal.github.com/en/actions/reference/workflows-and-actions/workflow-syntax. (Visited on 07/25/2025).

[80] *GitHub Event Types*. URL: https://docs-internal.github.com/en/rest/using-the-rest-api/github-event-types?apiVersion=2022-11-28. (Visited on 07/29/2025).

[81] *Issues Triaging*. URL: https://github.com/microsoft/vscode/wiki/Issues-Triaging. (Visited on 07/29/2025).

[82] Sun, Guoheng et al. *CoIn: Counting the Invisible Reasoning Tokens in Commercial Opaque LLM APIs*. May 2025. DOI: 10.48550/arXiv.2505.13778. arXiv: 2505.13778 [cs]. (Visited on 07/19/2025).

[83] Hilton, Michael et al. "Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. Paderborn Germany: ACM, Aug. 2017, pp. 197–207. DOI: 10.1145/3106237.3106270. (Visited on 07/27/2025).

A. Appendix

A.1. Source Code and Data

The source code of the thesis is available on GitHub split in two repositories. Both repositories have Readme files that explain how to use the code and data.

- **Prototype Repository:** Contains the implementation of the APR Core and the CI configuration. It is available at: https://github.com/justingebert/bugfix-ci
- **Evaluation Repository:** Contains the evaluation scripts and results of the APR system, including the QuixBugs benchmark setup. It is available at: https://github.com/justingebert/quixbugs-apr

A.2. LLM Versions

The following Table lists the exact LLM versions that where used during development and evaluation.

Table A.1.: *LLM models and their versions used in the system*

Model Name	**Version**
gemini-2.0-flash-lite	gemini-2.0-flash-lite
gemini-2.0-flash	gemini-2.0-flash
gemini-2.5-flash-lite	gemini-2.5-flash-lite-preview-06-17
gemini-2.5-flash	gemini-2.5-flash
gemini-2.5-pro	gemini-2.5-pro
gpt-4.1-nano	gpt-4.1-nano-2025-04-14
gpt-4.1-mini	gpt-4.1-mini-2025-04-14
gpt-4.1	gpt-4.1-2025-04-14
o4-mini	o4-mini-2025-04-16
claude-3-5-haiku	claude-3-5-haiku-20241022
claude-3-7-sonnet	claude-3-7-sonnet-20250219
claude-sonnet-4-0	claude-sonnet-4-20250514

AI Usage Card for Bachelor Thesis

PROJECT DETAILS	PROJECT NAME Bachelor Thesis	DOMAIN University Project	KEY APPLICATION Software Development
CONTACT(S)	NAME(S) Justin Gebert	EMAIL(S) justin.gebert.berlin@gmail.com	AFFILIATION(S) HTW Berlin
MODEL(S)	MODEL NAME(S) ChatGPT Github Copilot Gemini	VERSION(S) o3, 4.5, 4o latest 2.5 pro	

IDEATION	GENERATING IDEAS, OUTLINES, AND WORKFLOWS	IMPROVING EXISTING IDEAS	FINDING GAPS OR COMPARE ASPECTS OF IDEAS ChatGPT
LITERATURE REVIEW	FINDING LITERATURE ChatGPT Gemini	FINDING EXAMPLES FROM KNOWN LITERATURE OR ADDING LITERATURE FOR EXISTING STATEMENTS ChatGPT Gemini	COMPARING LITERATURE ChatGPT Gemini
METHODOLOGY	PROPOSING NEW SOLUTIONS TO PROBLEMS	FINDING ITERATIVE OPTIMIZATIONS	COMPARING RELATED SOLUTIONS
EXPERIMENTS	DESIGNING NEW EXPERIMENTS	EDITING EXISTING EXPERIMENTS	FINDING, COMPARING, AND AGGREGATING RESULTS
WRITING	GENERATING NEW TEXT BASED ON INSTRUCTIONS	ASSISTING IN IMPROVING OWN CONTENT OR PARAPHRASING RELATED WORK ChatGPT Github Copilot Gemini	PUTTING OTHER WORKS IN PERSPECTIVE Gemini
PRESENTATION	GENERATING NEW ARTIFACTS	IMPROVING THE AESTHETICS OF ARTIFACTS ChatGPT Github Copilot	FINDING RELATIONS BETWEEN OWN OR RELATED ARTIFACTS
CODING	GENERATING NEW CODE BASED ON DESCRIPTIONS OR EXISTING CODE	REFACTORING AND OPTIMIZING EXISTING CODE ChatGPT Github Copilot Gemini	COMPARING ASPECTS OF EXISTING CODE ChatGPT Github Copilot Gemini
DATA	SUGGESTING NEW SOURCES FOR DATA COLLECTION	CLEANING, NORMALIZING, OR STANDARDIZING DATA	FINDING RELATIONS BETWEEN DATA AND COLLECTION METHODS

ETHICS	WHY DID WE USE AI FOR THIS PROJECT? Consistency Efficiency / Speed	WHAT STEPS ARE WE TAKING TO MITIGATE ERRORS OF AI? All outputs were manually reviewed and verified.	WHAT STEPS ARE WE TAKING TO MINIMIZE THE CHANCE OF HARM OR INAPPROPRIATE USE OF AI? AI was used solely for enhancement, not for generating original content.

THE CORRESPONDING AUTHORS VERIFY AND AGREE WITH THE MODIFICATIONS OR GENERATIONS OF THEIR USED AI-GENERATED CONTENT

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt durch meine Unterschrift, dass ich die vorstehende Arbeit selbstständig und ohne fremde Hilfe angefertigt und alle Stellen, die ich wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen habe, als solche kenntlich gemacht habe, mich auch keiner anderen als der angegebenen Literatur oder sonstiger Hilfsmittel bedient habe. Die Arbeit hat in dieser oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen.

Angaben zur Verwendung KI-basierter Hilfsmittel

Im Anhang meiner Arbeit habe ich sämtliche KI-basierte Hilfsmittel angegeben. Diese sind mit Produktnamen und Anwendung in einem KI-Verzeichnis ausgewiesen. Ich versichere, dass ich keine KI-basierten Tools verwendet habe, deren Nutzung der Prüfer / die Prüferin explizit schriftlich ausgeschlossen hat. Ich bin mir bewusst, dass die Verwendung von Texten oder anderen Inhalten und Produkten, die durch KI-basierte Tools generiert wurden, keine Garantie für deren Qualität darstellt. Ich verantworte die Übernahme jeglicher von mir verwendeter maschinell generierter Passagen vollumfänglich selbst und trage die Verantwortung für eventuell durch die KI generierte fehlerhafte oder verzerrte Inhalte, fehlerhafte Referenzen, Verstösse gegen das Datenschutz- und Urheberrecht oder Plagiate. Ich versichere zudem, dass in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt.

_____

Datum, Ort, Unterschrift