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

Danksagung

[Text der Danksagung]

Abstract

Generative AI is reshaping software engineering practices by automating more tasks every day, including code generation, debugging and program repair. Despite these advancements, existing Automated Program Repair (APR) systems frequently suffer from complexity, high computational demands, and missing integration within practical software development lifecycles. Such shortcomings often lead to frequent context switching, which negatively impacts developer productivity.

In this thesis, we address these challenges by introducing a novel and lightweight Automated Bug Fixing system leveraging LLMs, explicitly designed for seamless integration into CI/CD pipelines deployed in budget constrained environments. Our containerized approach, developed with a strong emphasis on security and isolation, manages the complete bug-fixing lifecycle from issue creation on GitHub to the generation and validation of pull requests. By automating these processes end-to-end, the system significantly reduces manual intervention, streamlining developer workflows and enhancing overall productivity.

We evaluate our APR system using the QuixBugs benchmark, a recognized dataset for testing APR methodologies. The experimental results indicate that our streamlined and cost-effective solution effectively repairs small-scale software bugs, demonstrating practical applicability within typical software development environments.

The outcomes underscore the feasibility and advantages of integrating APR directly into real-world CI/CD pipelines. We also discuss limitations inherent in LLM-based solutions, such as accuracy and reliability issues and suggest future enhancement and research.

-add that this approach directly integrates into the development lifecycle reducing configuration ... - add peak of results - add that multiple models have been tested

Contents

List of Figures

List of Tables

Listings

List of Abbreviations

AI	Artificial Intelligence
ML	Machine Learning
API	Application Programming Interface
LLM	Large Language Model
JSON	JavaScript Object Notation
UI	User Interface
SLDC	Software Development Lifecycle
CI	Continuous Integration
GenAI	Generative AI
APR	Automated Program Repair

1. Introduction

Generative AI is rapidly changing the software industry and how software is developed and maintained. The emergence of Large Language Models (LLMs), a subfield of Generative AI, has opened up new opportunities for enhancing and automating various domains of the software development lifecycle. Due to their remarkable capabilities in understanding and generating code, LLMs have become valuable tools for developers. Everyday tasks such as requirements engineering, 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 can harm software quality by causing crashes, vulnerabilities, or even data loss [5]. In fact, according to 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 research interest. 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. Continuous Integration 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 cost-effective prototype for automated bug fixing that seamlessly integrates with Continuous Integration (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 in terms

of repair success rate, cost-effectiveness and performance?

The thesis is organized as follows:

Section 2 provides theoretical background on Software Development, Generative AI in the context of software development, and Automated Program Repair.
Section 3 outlines the methodology used for this thesis, consisting of preparation, implementation, and evaluation.
Section 4 showcases the functional and non-functional requirements constructed for the prototype.
Section 5 explains the implementation and resulting system in detail.
Section 6 showcases the resulting workflow when using the prototype and lists the results of the evaluation.
Section 7 discusses the potentials and limitations of the prototype based on the results.
Finally, section 8 concludes the thesis by summarizing the findings and contributions of this work.

2. Background and Related Work

In this section, the required theoretical background and context for this thesis are explained. First, fundamental concepts of software development, the agile software development lifecycle (SDLC), Continuous Integration (CI), and software project hosting platforms are introduced. The second part explores Generative AI and LLMs with their rising role in software development practices. The third part examines the evolution and current state of Automated Program Repair (APR) with examples of existing approaches.

2.1. Software Development

The following section introduces core concepts of software development, starting with the software development lifecycle, followed by the importance of Continuous Integration (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, software development lifecycle models have been introduced. These frameworks constantly evolve to adapt to the changing needs of software creation. One of the most promising and widely used models today is Agile [20, 21].

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

An Agile iteration consists of multiple stages. Figure 2.1 shows 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].

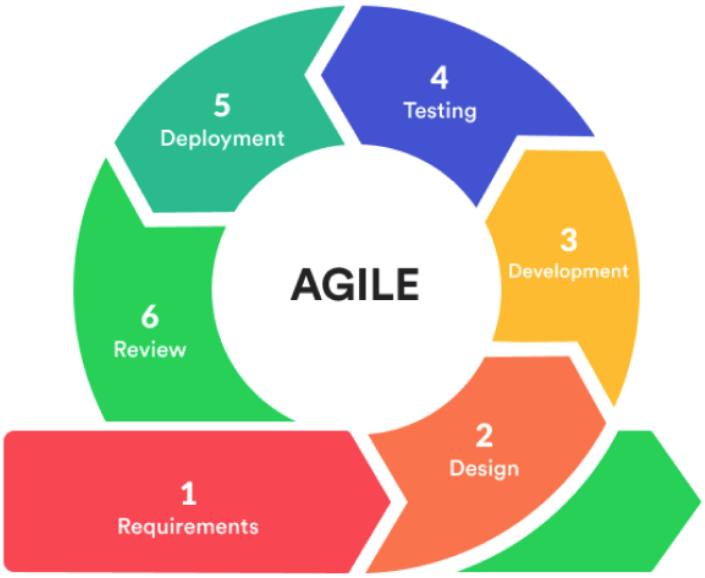

Figure 2.1.: *Agile software development lifecycle*

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 that 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 [24]. 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 [25, 4].

2.1.2. Continuous Integration

Continuous Integration (CI) has become a standard practice in agile software development to accelerate software development and delivery. CI enables frequent code integration into a repository, 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, with a report generated for the developer.

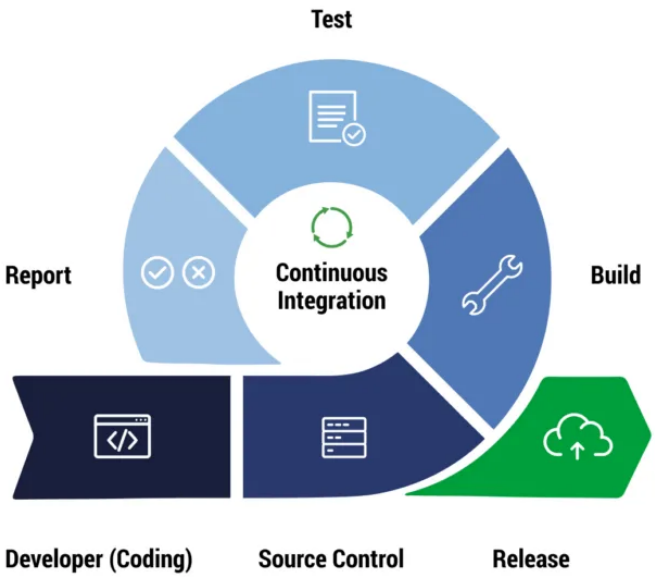

Figure 2.2.: *Continuous Integration cycle*

Although Continuous Integration improves feedback loops, it can also add overhead. Effort and infrastructure must be invested to keep pipelines running [26], and projects may suffer from long build times that harm developer productivity [27].

2.1.3. Software Project Hosting Platforms

Modern software projects live on platforms like GitHub or GitLab. GitHub alone hosts 100 million developers and more than 518 million repositories, making it the largest open-source community worldwide [28].

Development platforms offer tools and services for the entire software development lifecycle, including project hosting, version control, issue tracking, bug reporting, project management, backups, collaborative workflows, and documentation capabilities [29, 21].

GitHub Issues are a key feature of GitHub, this feature enables project-scoped backlogs and tracking of tasks, features and bugs. 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 [30]. Figure 2.3 shows an example of a GitHub Issue.

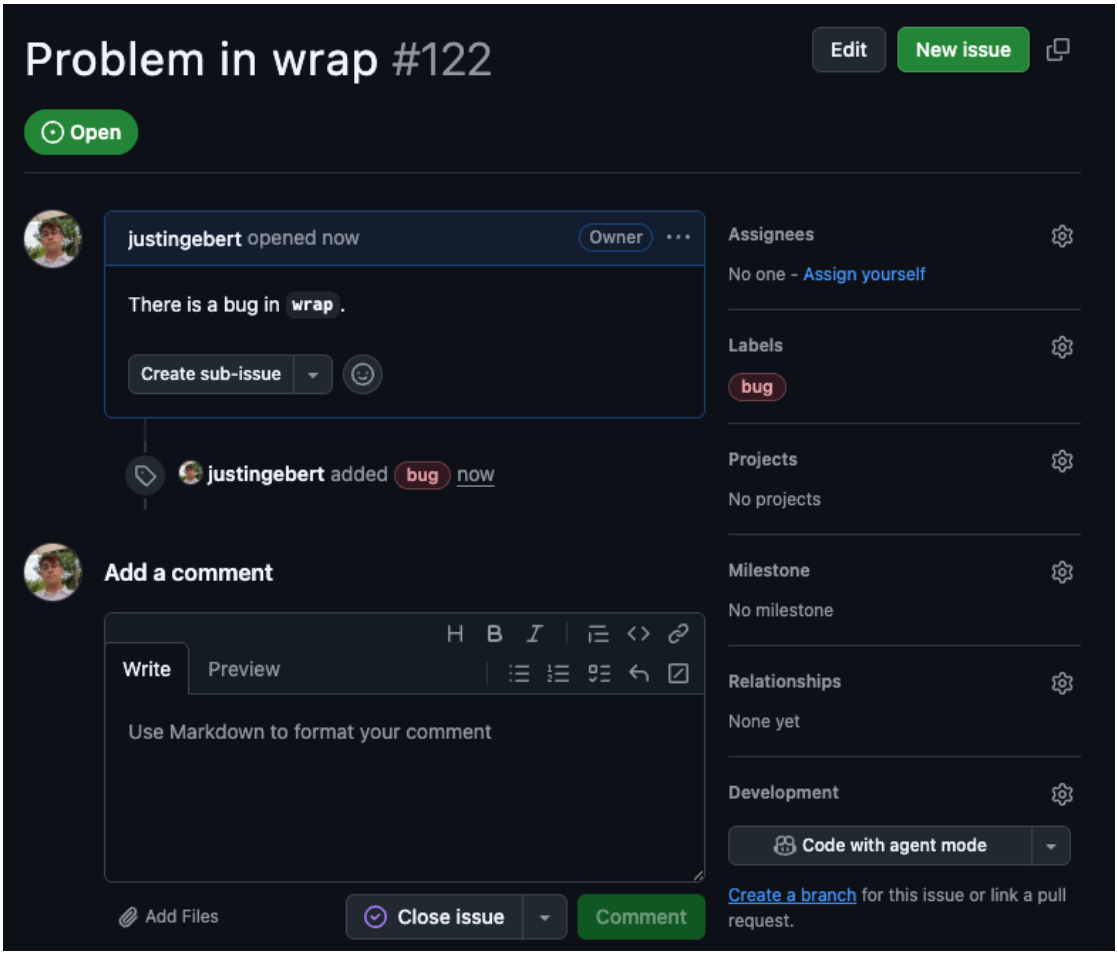

Figure 2.3.: *Example of a GitHub Issue*

For integrating and reviewing code, GitHub uses 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 [31].

Moreover, GitHub also provides a managed solution (GitHub Actions) for running CI pipelines in repositories. CI pipelines are configured by writing workflow files in YAML. Workflows can run self-hosted or hosted by GitHub. A workflow consists of triggers, jobs, and steps. One or more events trigger a workflow, which executes one or more jobs consisting of multiple steps [32]. Figure 2.4 visualizes part of a workflow on GitHub.

[1] TODO explain format

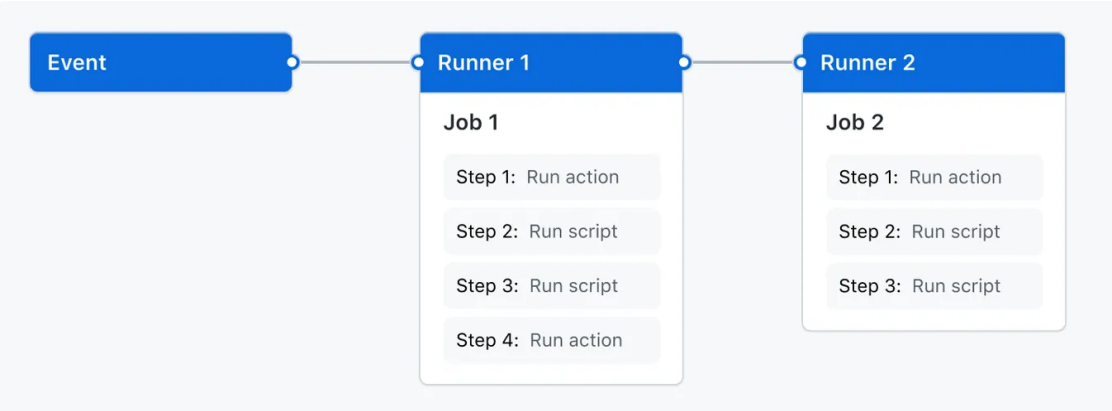

Figure 2.4.: *Components of a GitHub Action*

Workflow results and logs can be viewed from multiple places in the GitHub Web User Interface (UI), including the Actions tab, the 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 their repositories [33].

2.2. Generative AI in Software Development

This section covers the role of Generative AI in modern software development. First, Generative AI and Large Language Models (LLMs) are defined. The second part focuses on the impact of Generative AI on software development practices.

2.2.1. Generative AI and Large Language Models

Generative Artificial Intelligence (GenAI) is a subfield of artificial intelligence referring to systems that generate new content based on patterns learned from extensive training data. Advanced machine learning techniques, particularly deep learning, enable these systems to generate text, images, or code that resembles human-generated output [34].

The introduction of the transformer architecture revolutionized the field of text generation and natural language processing (NLP). This architecture laid the groundwork for Large Language Models [35, 36]. Extensive training results in LLMs with billions of parameters, allowing them to understand and generate text in natural languages and diverse programming languages. Research has shown that a model's size impacts its performance, with larger models generally achieving better results in various NLP tasks [37]. However, training and operating larger models require significant computational resources [38, 36]. Furthermore, despite modern LLMs showing promising results in text generation, they can still hallucinate incorrect or biased content [38].

To achieve specific tasks using LLMs, an input prompt needs to be provided to the model. Designing these inputs to guide a model's output is called prompt engineering. This process is crucial for achieving desired results from LLMs, as the quality and specificity of the prompt directly influence the model's output. Text used for input

and output of LLMs 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 [36].

Large Language Models can be accessed via APIs offered by providers like OpenAI, Anthropic, or Google. A selection of LLMs with characteristics is shown in section 3.1.2.

2.2.2. Large Language Models in Software Development

Large Language Models are reshaping software development by automating various tasks [1]. With billions of parameters and pre-training on massive codebases, these models exhibit extraordinary capabilities in this area [18]. Tools like ChatGPT[2] and GitHub Copilot[3] have become popular in the software development community, providing developers with AI-powered code suggestions and completions [39]. These tools are applied in various stages of the software development lifecycle, including requirements engineering, code generation, refactoring, testing, and debugging [1, 2, 39]. By using LLMs, development cycle times can be reduced by up to 30 percent [39, 40]. Furthermore, these tools positively impact developer satisfaction and reduce cognitive load [40].

Despite the rapid adoption of Generative AI in many areas of software development, this technology still faces limitations. LLMs have difficulty with tasks outside their training scope or requiring specific domain knowledge [1]. Limited context windows create challenges when working with large codebases and complex projects, restricting true contextual or business requirement understanding [39]. When generating code, LLMs can produce incorrect or insecure outputs, leading to additional bugs and vulnerabilities [1, 39]. Additionally, integrating LLMs can introduce vulnerabilities to prompt injection, where malicious instructions lead to harmful code generation [41]. Moreover, code generated by LLMs is based on existing training data, raising questions about ownership, responsibility, and intellectual property rights [42, 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 these approaches in more detail.

Recent research is exploring solutions integrating LLMs into existing software development practices and workflows, leveraging existing development tools and platforms for seamless integration into the software development lifecycle [2, 43, 44, 42].

[2] link to gpt
[3] link to copilot

2.3. Automated Program Repair

Automated Program Repair (APR) describes software used to detect and repair bugs in codebases with minimal human intervention [45]. APR aims 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: first localizing the bug, then repairing it, and finally validating the fix [45, 46]. 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 [46] is a prominent example of an APR system applied at scale, used at Meta to automatically fix common bugs in their production codebases.

The field of APR has greatly benefited from rapid advancements in AI and ML, 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 common APR benchmarks used in research and industry.

2.3.1. Evolution of Automated Program Repair

Automated Program Repair 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, 47, 48]. 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 [49].
- **Constraint/Semantic-based repair** synthesizes patches using constraint solvers derived from the program's semantic information and test cases [12, 50]. Angelix is a prominent example of this approach [50].
- **Template-based repair** relies on mined templates for transformations of known bugs [47]. These templates are mined from previous human-developed bug fixes [47, 48]. GetaFix is an industrially deployed tool, learning recurring fix patterns from past fixes [46].

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, 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 [47, 51]. Prominent examples include CoCoNut [52] and Recoder [53]. Despite significant advancements, these methods remain limited by training data and struggle with unseen bugs [54].

The Emergence of LLM-based APR:

The recent rapid 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 [55]. Consequently, LLMs form the foundation of a new APR paradigm [18, 56].

Different LLM-based approaches have emerged and are actively researched, categorized into four main paradigms:

- **Retrieval-Augmented 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, 48].
- **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 [56, 2, 57]. 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 [43].
- **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, 57].

Popular LLMs for APR include ChatGPT, Codex, CodeLlama, DeepSeek-Coder, and CodeT5 [1, 48, 56]. 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 [58, 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 [56]. 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 2.1 [59].

Model	**Languages**	**Number of Bugs**	**Description**
QuixBugs [60]	Python, Java	40	small single line bugs
Defects4J [61]	Java	854	real-world Java bugs
ManyBugs [62]	C	185	real-world C bugs
SWE Bench [63]	Python	2294	Real GitHub repository defects
SWE Bench Lite	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 Continuous Integration. We aim to answer the following research questions to evaluate the system's capabilities and impact on the software development process:

- **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 in terms of repair success rate, cost-effectiveness and developer workflow enhancement?

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

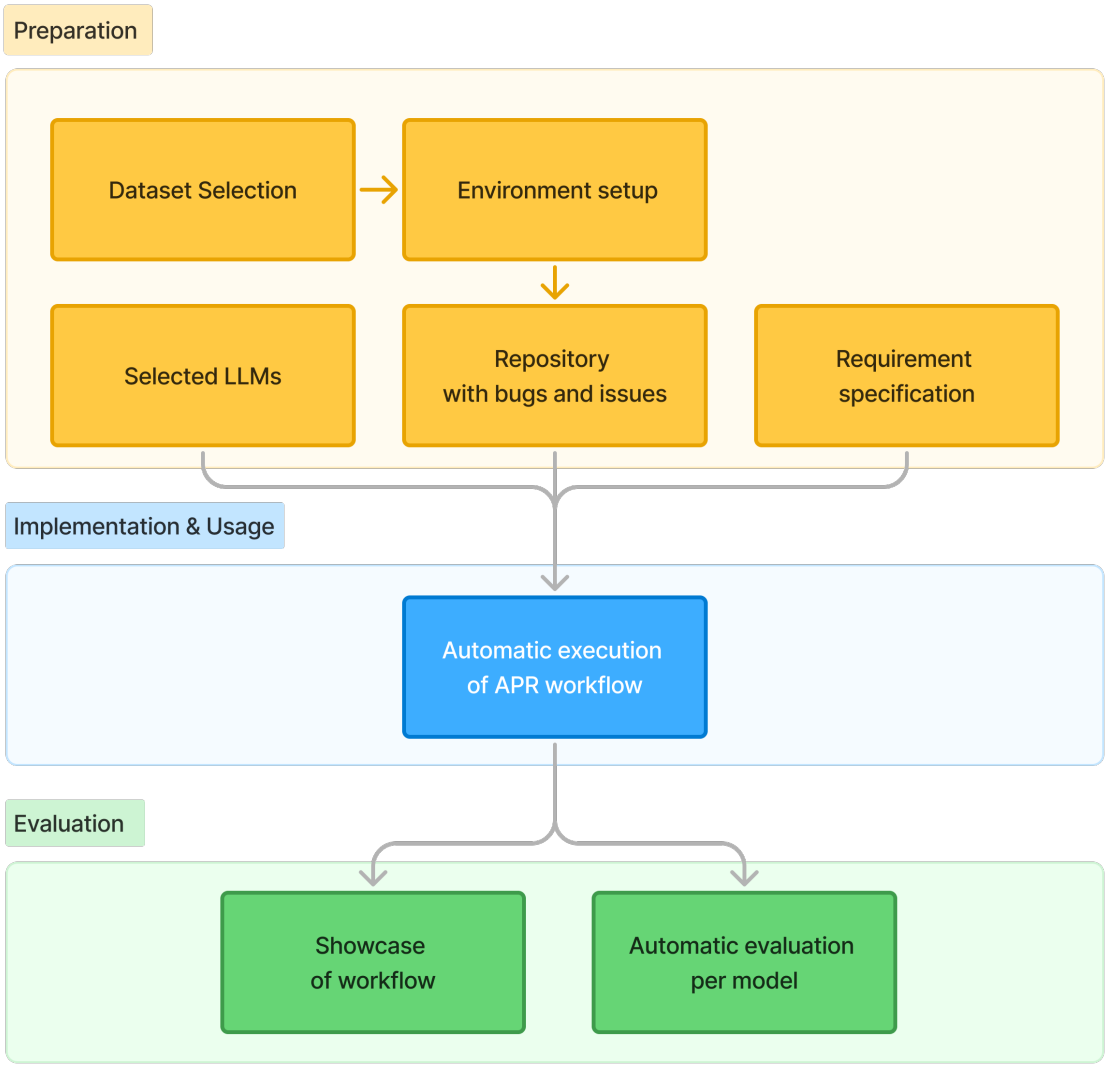

Figure 3.1.: *Thesis methodology approach*

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. By specifying requirements, we lay the groundwork for the implementation of the APR system. In the second part, we implement the APR system as a GitHub Action workflow based on the requirements. Lastly, we evaluate the self-developed prototype using the defined evaluation metrics 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 first need to prepare an environment where the system can be integrated, used, and evaluated. This includes selecting

a suitable dataset and Large Language Models, 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 [60]. This dataset is well-suited for our purposes due to its focus on small-scale bugs in Python[1]. It consists of 40 individual files, each containing an algorithmic bug. Each bug is 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 [60], enabling us to evaluate whether our system can take over the cognitively demanding task of fixing small bugs without developer intervention.

Additionally, compared to other APR benchmarks 2.1 like SWE-Bench [63], QuixBugs is relatively small, which allows for accelerated setup and development.

3.1.2. LLM Selection

For the evaluation of our APR system, we will test a selected pool of LLM models. We evaluate with the latest models released before 11 July 2025 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 and cost-effectiveness of the APR system. Table 3.1 shows the selected models that will be used for evaluation with the following data:

- (1) Model Name: The name of the LLM model.
- (2) Publisher: The company or organization that developed the model.
- (3) Context Window Size in Tokens: The maximum number of tokens the model can process in a single request.
- (4) Cost per 1M Tokens: The cost of processing 1 million tokens, divided into input and output cost.
- (5) Provider Description: Description of the model's characteristics.

Table 3.1.: *Characteristics of selected LLMs (21.07.2025)*

(1)	(2)	(3)	(4)	(5)
gemini-2.0-flash-lite	Google	1,048,576	input: $0.075 output: $0.30	X
gemini-2.0-flash	Google	1,048,576	input: $0.15 output: $0.60	X
gemini-2.5-flash-preview	Google	1,000,000	input: $0.10 output: $0.40	X

[1]The QuixBugs benchmark contains the same bugs translated to Java as well. We will exclude this for evaluation research

gemini-2.5-flash	Google	1,048,576	input: $0.30 output: $2.50	X
gemini-2.5-pro	Google	1,048,576	input: $1.25 output: $10.00	X
gpt-4.1-nano	OpenAI	1,047,576	input: $0.10 output: $0.40	X
gpt-4.1-mini	OpenAI	1,047,576	input: $0.40 output: $1.60	X
gpt-4.1	OpenAI	1,047,576	input: $2.00 output: $8.00	X
o4-mini	OpenAI	200,000	input: $1.10 output: $4.40	X
claude-3-5-haiku	Anthropic	200,000	input: $0.80 output: $4.00	X
claude-3-7-sonnet	Anthropic	200,000	input: $3.00 output: $15.00	X
claude-sonnet-4-0	Anthropic	200,000	input: $3.00 output: $15.00	X

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 relevant 40 files, we generate a GitHub issue for each bug. 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.

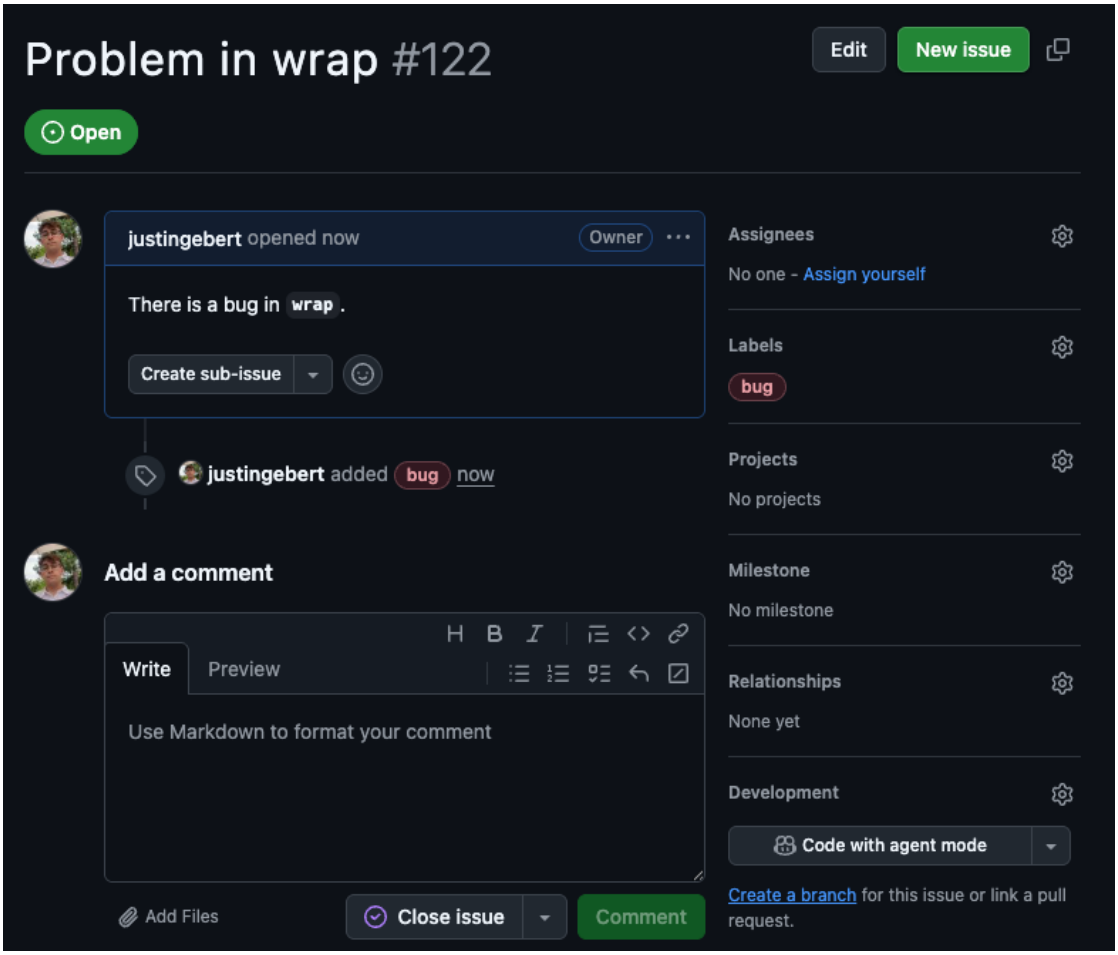

Figure 3.2.: *Example of a generated GitHub Issue*

3.1.4. Requirements Specification

Before the implementation phase, we constructed requirements for the prototype, applying the INVEST model, a widely adopted method in Agile software development for engineering requirements [64]. 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. The resulting requirements are detailed in 4.

3.2. System Implementation

In this section, we provide a high-level overview of the implemented Automated Bug Fixing Pipeline. A detailed implementation description can be found in 5.

The Automated Bug Fixing Pipeline was developed using iterative prototyping and testing, with a focus on simplicity and extensibility. Based on the self-developed

requirements 4, we built the system visualized in Figure 3.3 and Figure 3.4.

The integration of the pipeline into the repository is performed during this implementation phase. Once the system is in place, the 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 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 and providing details about the repair process. In case of an unsuccessful repair or exhaustion of the maximum number of attempts, a failure is reported to the issue.

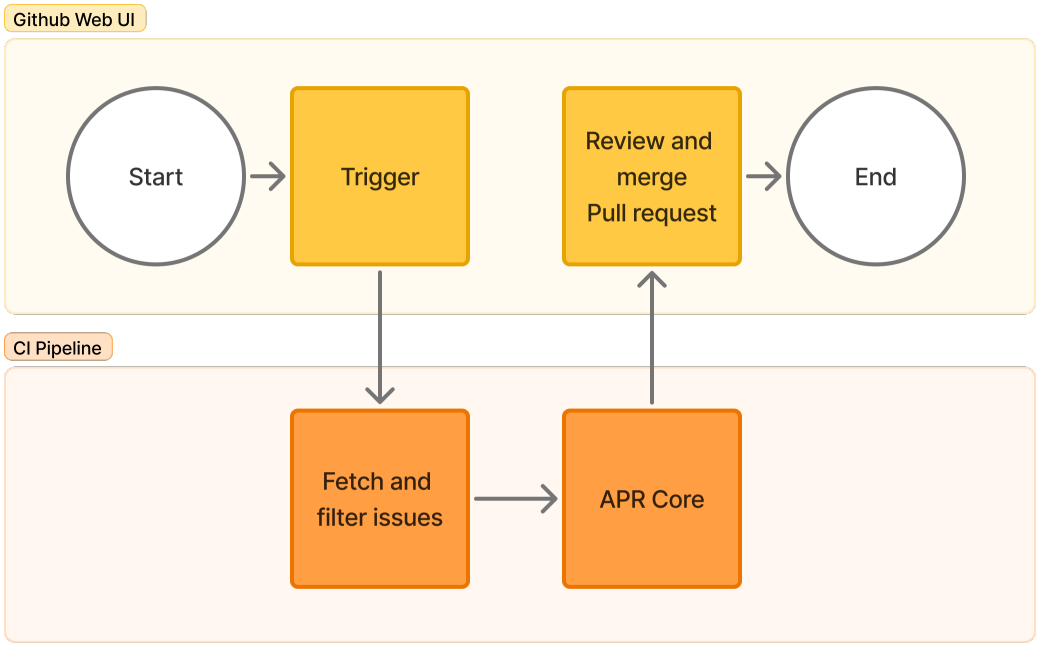

Figure 3.3.: *Overview of the APR Pipeline*

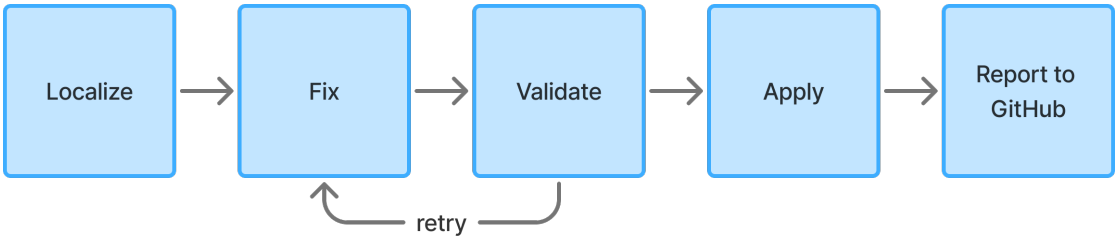

Figure 3.4.: *Overview of the APR Core*

3.3. Evaluation

In this section, we describe how we measure the effectiveness, performance, and cost of the APR pipeline when integrated into a GitHub repository. We use GitHub Actions' Continuous Integration capabilities in combination with the prepared QuixBugs repository as a base. All selected LLM models from table 3.1.2 are evaluated in the context of the APR system. The evaluation is based on data collected during and after the execution of the APR pipeline.

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

To evaluate the effectiveness of the LLM models and the approach, we determine a repair success rate. A successful repair of an issue is determined by validation and the complete test suite provided by QuixBugs. An issue is considered successfully repaired when the validation passes with correct syntax and the related test suite passes all tests.

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

To assess performance, we analyze collected fine-grained 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 run minutes are not included in the cost estimation, as 2000 minutes are included in the free tier of GitHub for public repositories.

The following data is automatically collected by each run of the APR system. Using self-developed scripts, we collect the data from different sources for each run. Below, we list the relevant data collected for each run 3.2, each issue processed 3.3, and each stage of an issue repair process 3.4. We use the collected data to calculate results for different runs testing each of the selected models, which will help in answering RQ2. The calculations are listed in Table 3.5.

Metric	**Description**	**Source**
Run ID	Unique identifier for the workflow run	Github Action Runner
Configuration	Configuration details, including LLM Model used and max attempts	Github repository / APR Core
Execution Time	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 3.2.: *Summary of metrics collected for each run*

Metric	Description	Source
Issue ID	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 3.3.: *Summary of metrics collected for each processed issue*

Metrics	Description	Source
Stage ID	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 3.4.: *Summary of metrics collected for each stage*

Metrics	Calculation
Repair Success Rate	$\frac{NumberofSuccessfulRepairs}{TotalIssuesProcessed}$
Average Execution Time per Issue	$\frac{TotalExecutionTime}{TotalIssuesProcessed}$
Average Cost per Issue	$\frac{TotalCost}{TotalIssuesProcessed}$
APR Core Execution Time	$\frac{TotalAPRCoreExecutionTime}{TotalIssuesProcessed}$
CI Pipeline Execution Time	$\frac{TotalCIPipelineExecutionTime}{TotalIssuesProcessed}$

Table 3.5.: *Run evaluation metrics calculated from the collected data*

4. Requirements

To guide the implementation of the prototype, we developed the following requirements to help design the system. These requirements enable better planning and prioritization during development and allow for tracking progress throughout the implementation. We constructed both functional 4.1 and non-functional 4.2 requirements.

4.1. Functional Requirements

Table 4.1.: *Functional requirements*

ID	Title	Description	Verification
F0	Multi Trigger	The Pipeline can be triggered: manually, scheduled via cron or by issue creation/labeling.	Runs can be found for these triggers
F1	Issue Gathering	Retrieve GitHub repository issues and filter them for correct state and configured labels `BUG`.	`gate` logs list of fetched issues.
F2	Code Checkout	Fetch the repository code into a fresh workspace and branch (via Docker mount).	After F2, the Core has access the source files.
F3	Issue Localization	Use LLM to analyze the issue description and identify relevant files.	LLM output contains file paths with files that shall be edited.
F4	Fix Generation	Use LLM to edit the identified files.	LLM output contains adjusted content for the identified files.
F5	Change Validation	Run format, lint and relevant tests and capture pass/fail status.	Context shows build and test results.
F6	Iterative Patch Generation	If F5 reports failures, retry F4-F5 up to `max_attempts` times.	Multiple stage execution are shown in context when Validation fails
F7	Patch Application	Commit LLM-generated edits to the issue branch.	Git shows a new branch with a commit referencing the Github issue

F8	Result Reporting	Report using Pull Requests and issue comments.	A PR or appears for each issue, showing diff and summary.
F9	Log and Metric Collection	Provide log files and Metrics for evaluation and debugging.	Log and metric files are accessible

4.2. Non-Functional Requirements

Table 4.2.: *Non-Functional requirements*

ID	Title	Description	Verification
N1	Containerized Execution	All APR code runs in CI runner in a Docker container.	Workflow shows Docker container usage
N2	Configurability	User can specify issue labels, branches, attempts, LLM models.	Changing the config file alters agent behavior accordingly.
N3	Portability	The system can be deployed on any python repository on GitHub.	The system repairs at least one issue in more than one repository
N4	Reproducibility	Runs are deterministic given identical repo state and config.	Multiple runs on the same issue report similar metrics.
N5	Observability	The system provides logs and metrics.	Logs and metrics files are generated for each run.
N6	No Manual Intervention	The system runs fully automatically without user input.	Issue creation to fix Pull Request works without any manual steps.

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 sections. The full implementation can be found in the appendix A.

The resulting system consists of two main components. The **APR core** which holds the core logic for the repair process. The second component is the **Continuous Integration Pipeline** which integrates the core logic within a GitHub repository. It serves as the entry point and orchestrates the execution of the APR core based on configured triggers events of 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[1]. Embedding it into a Docker Image [2] makes it easy to deploy, portable and small in memory. In order to use the APR core the following data (displayed in table 5.1) needs to passed to the container:

Table 5.1.: *Container Inputs*

Name	Description	Type
Source Code	Git repository where APR fix bugs	Docker 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_REPO	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

[1] todo ask ZHANG do i need this
[2] link to docker

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 main data structure for the issue repair process and is used at every step. 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*

The context is passed between stages, with each stage performing a specific task in the bug fixing process and returning an updated context. The core stages are Localize, Fix, Build, and Test.

The repair process starts with the localization stage. This stage attempts to identify the files in the codebase required to fix the bug, using the configured LLM model via the provider's SDK[3]. The 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. The localization system instruction and prompt are shown in 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.

```

[3]explain

```
    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 the localized files in context, the Fix stage comes next. This stage again uses the configured LLM model API to generate a fix for the issue in the localized files. The prompt includes the issue details and file names along with their content. The response is expected to contain a list of edits for each file. The LLM may also specify that no changes are required in certain files. The generated response is then parsed and applied to the files in the workspace. Finally, the context is updated with the new file content. Below is the system instruction and base prompt used for the Fix stage 5.3.

```
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 the generated edits, two stages are available: Build and Test. The Build stage is responsible for validating the syntax of the changes made in the Fix stage. It checks if the code can be built. For Python code, this means checking if all syntax is valid and follows standardized code quality and maintenance rules [4]. To achieve this,

[4] todo

the code is first formatted using the Python formatter Black[5] and then linted using flake8[6]. This ensures properly formatted code and appends any warnings or errors to the context.

After validation, the generated code is tested, if a test command is configured. 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 and attached to the prompt. When the maximum number of attempts is reached and the code still does not pass testing, an unsuccessful repair is reported to the issue by creating a comment using the GitHub API.

If validation and testing are successful, the issue is marked as successfully repaired. The file changes are committed and pushed to the remote repository. 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 issue.

At this point, integration with the GitHub repository is completed. During execution, the APR Core logs every action, which can be used for debugging and makes the repair process more transparent. 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 system. A summary of the metrics is mentioned in 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 functionalities (blue) and tools (green) used in the APR Core.

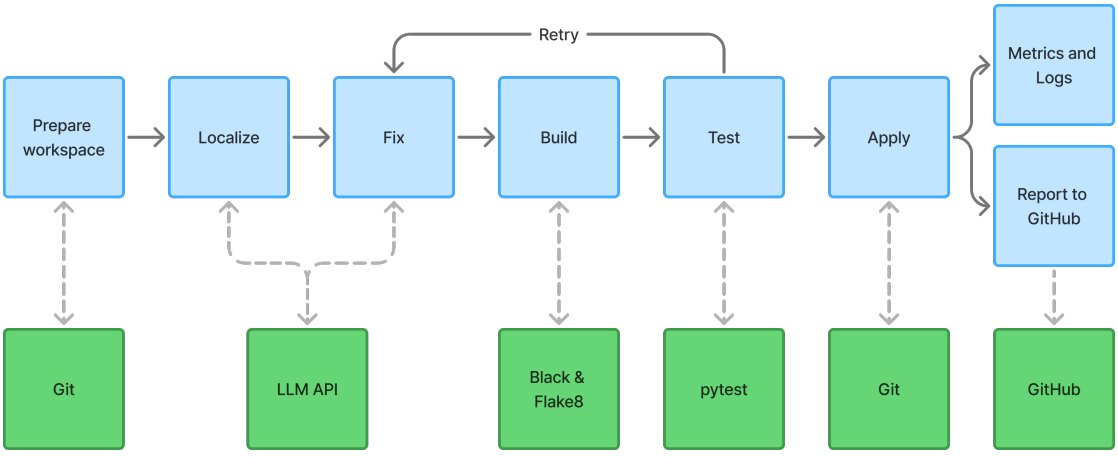

Figure 5.1.: *APR Core Logic*

[5] todo
[6] todo

Continuous Integration Pipeline:

A GitHub Action Workflow integrates the APR Core into the GitHub repository. The workflow is written in YAML[7] according to the GitHub Action standard[8]. For this prototype, we use Linux x64 runners[9] 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 multiple triggers and jobs. Triggers are based on events[10] 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 the event. The issue is passed from the ("issue_labeled") event when labeled with the configured labels or from (‘"issue_comment") when extra 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 data to determine if the issue should be processed or skipped using a Python script ("filter_issues.py"). This script must be placed where it is accessible to the workflow file. It checks the labels of the issue and evaluates its state to determine if it is relevant for the APR process.

If no issues pass the "gate", the "skipped" job is executed, which simply logs that no issues were found and exits the workflow run. If issues do pass the "gate", the "bugfix" job is started. This job is responsible for executing the APR Core logic. It sets up the necessary prerequisites (see 5.1) to start a container using the latest 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.

The final step of 'bugfix' is uploading the logs and metric files from the APR Core as artifacts, making them available after the workflow run has completed.

Figure 5.2 visualizes the workflow and its jobs.

[7] todo
[8] link to GitHub
[9] default runner, sufficient for this purpose
[10] explain and link

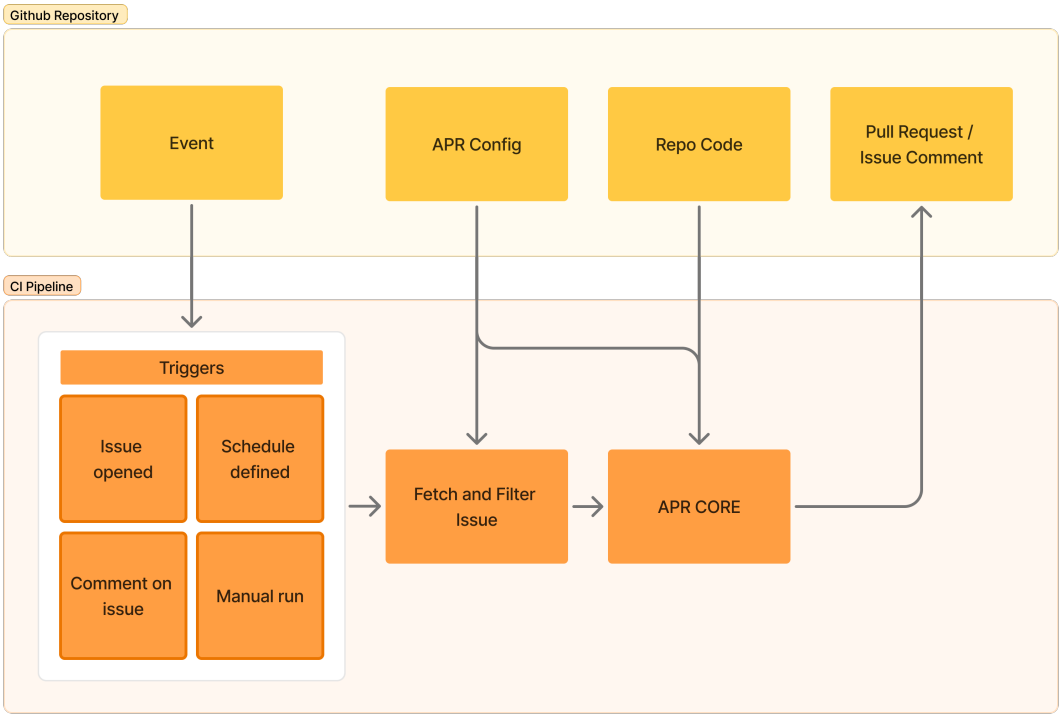

Figure 5.2.: *APR Core Logic*

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.

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 uses a default setup. A custom configuration must be named "bugfix.yml" and placed at the root of the repository. This setup allows for easy customization of the system without modifying the code. The configuration is read by system components during execution. 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 fails.
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 of the repository 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

The following sections outline how each requirement was satisfied.

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 Continuous Integration, 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. In the second part of this section, we present the results of the quantitative evaluation of the prototype being used on the repository containing the QuixBugs dataset as a basis.

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". With these in place, the system is ready to operate. Additionally, an optional configuration file can be added to the root. This '.bugfix.yml" file can be used to override the LLM model used, max attempts, and naming conventions for labels and branches. 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 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[1].

[1] explain triage

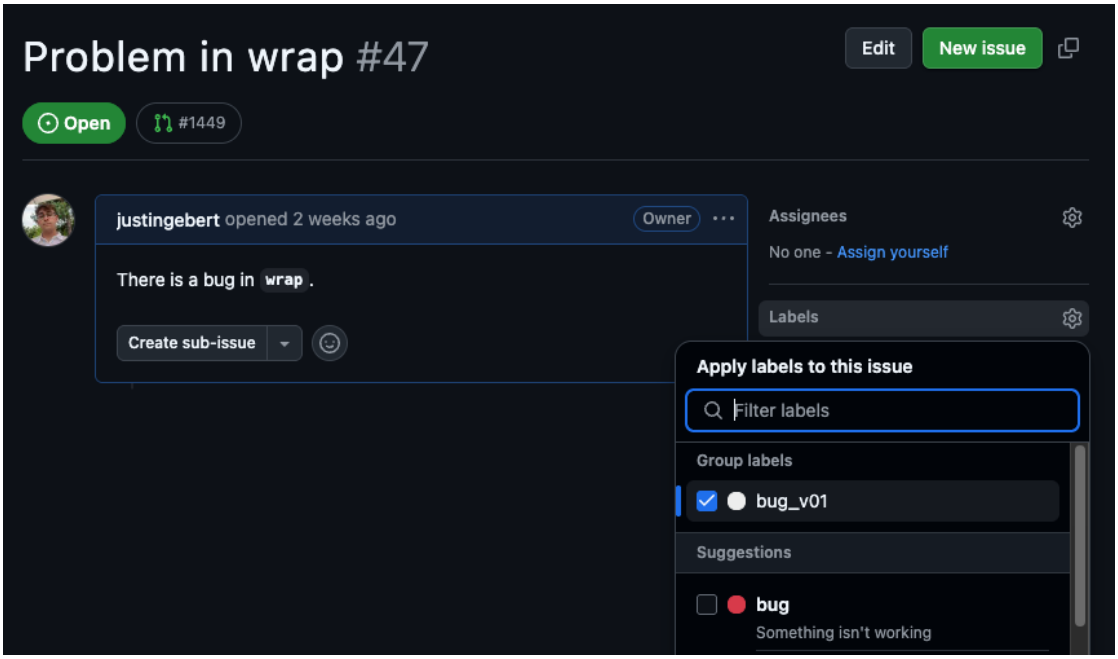

Figure 6.1.: *Trigger automatic fixing for single issue*

The second way collects and processes all issues labeled with the "bug_v01" label by scheduling the workflow to run at a specific time or dispatching it manually (see Figure 6.2). This allows for a more controlled approach to bug fixing, where issues are processed in batches.

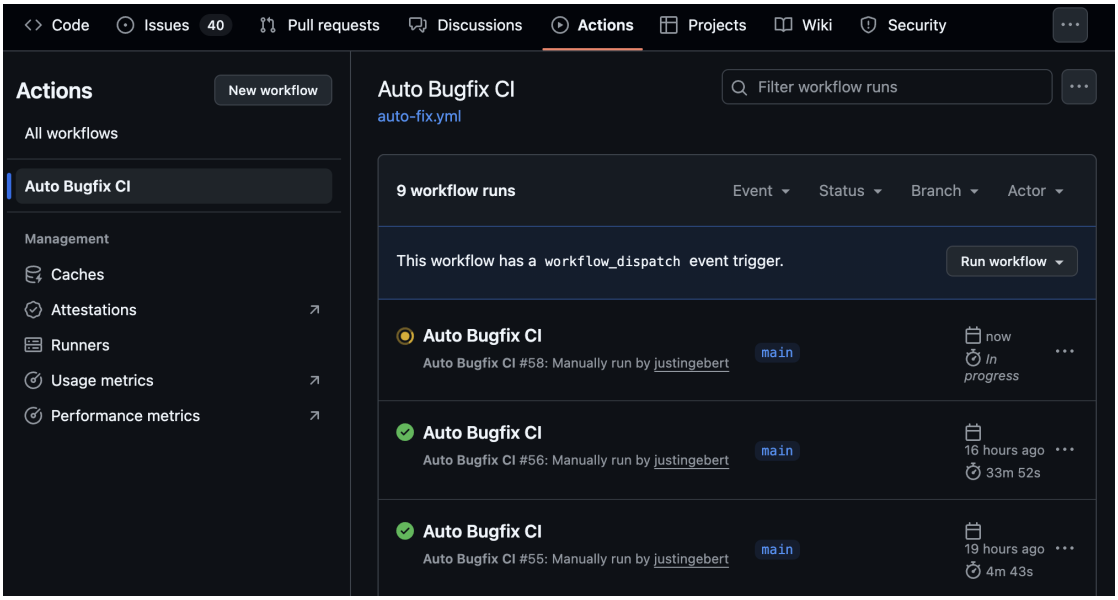

Figure 6.2.: *Manual Dispatch of APR*

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 5.

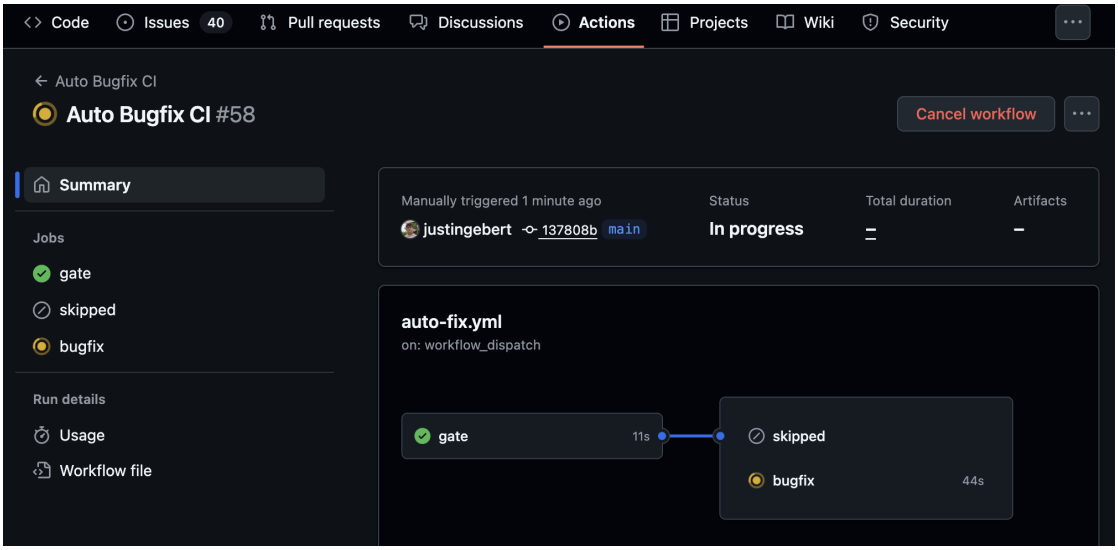

Figure 6.3.: *GitHub Action Run*

A run can produce two possible outcomes for each issue it processes. Firstly, on a successful repair attempt, it creates a Pull Request with the changes made to the codebase and links the issue to it. The created Pull Requests allow code review and merging of the fixed changes into the main branch. When merged, the issue will be automatically closed. Figure 6.4 shows an example of a resulting Pull Request created by the APR system.

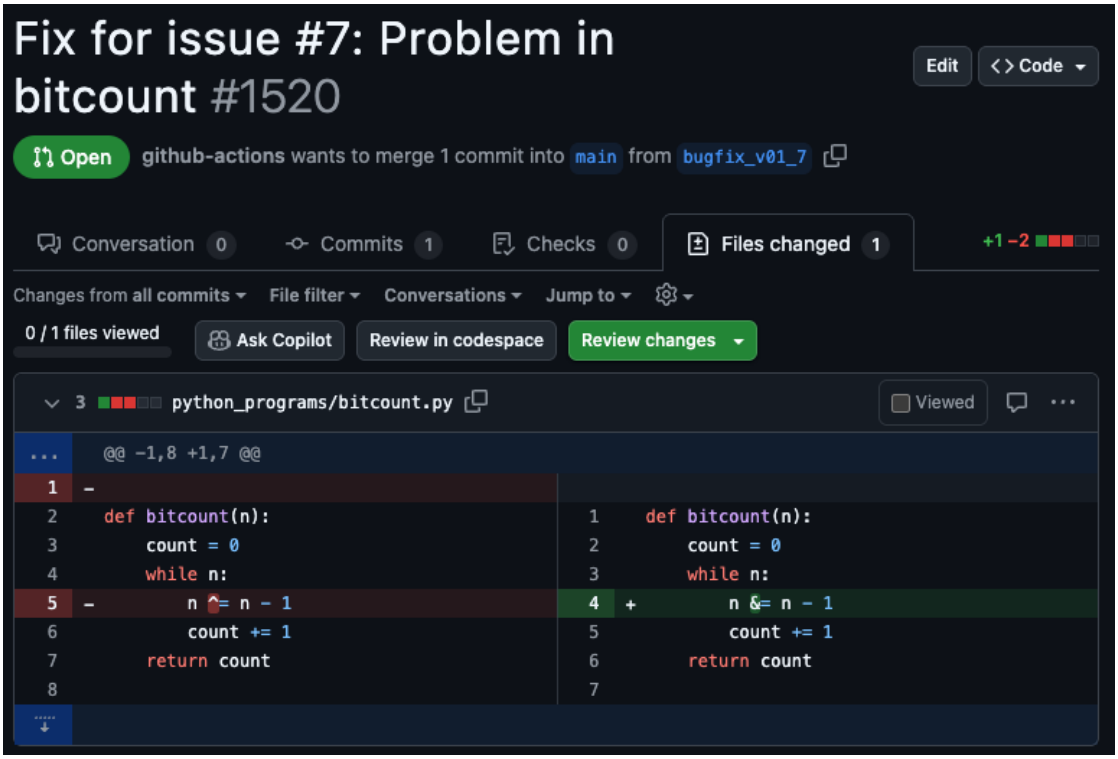

Figure 6.4.: *Resulting Pull Request*

Secondly, 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 (see Figure 6.5) so it won't be picked up again. This allows for easy tracking of issues that could not be fixed by the APR system.

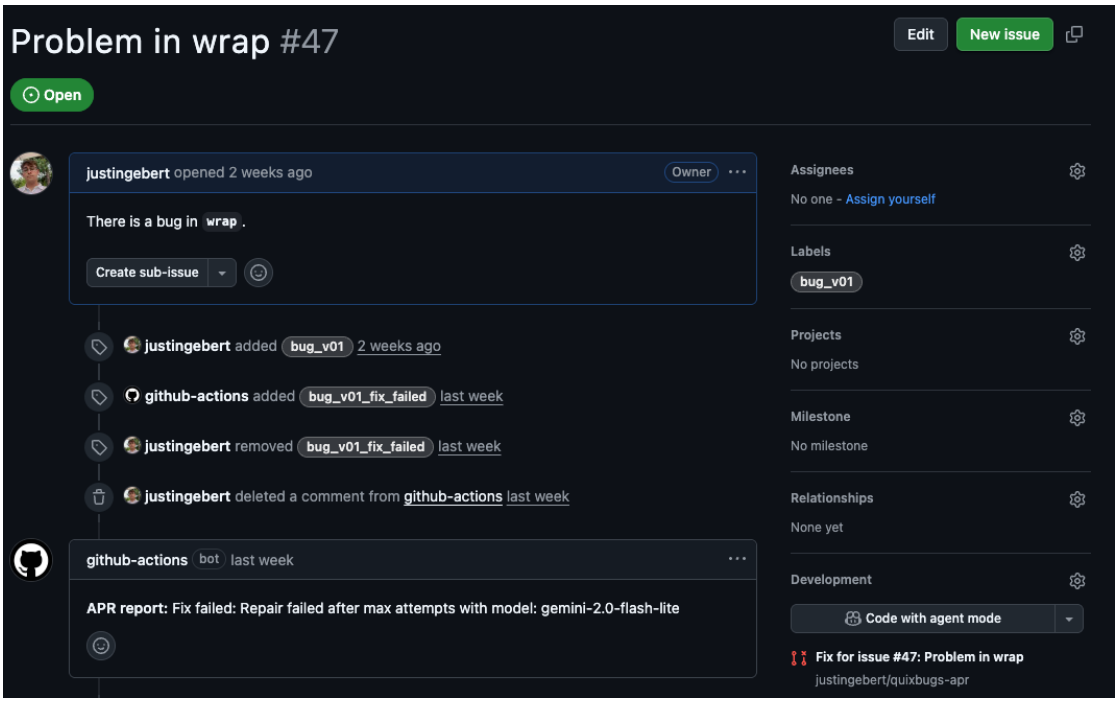

Figure 6.5.: *Failure Report*

Failed issues 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 new 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 are published as artifacts to each run, available to download in the run view.

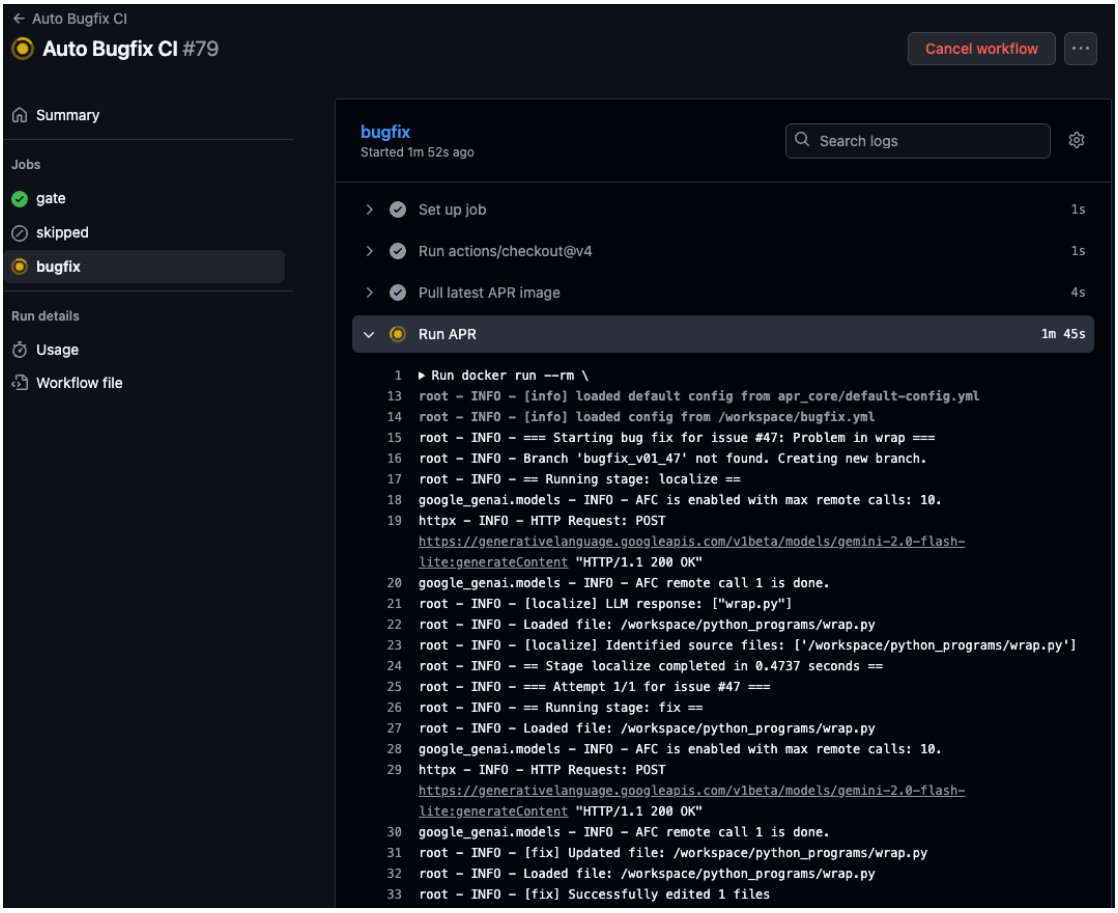

Figure 6.6.: *APR log stream*

Figure 6.7 visualizes the resulting flow of the APR system. The diagram shows the relation between user actions (yellow) and runs by the integrated APR system.

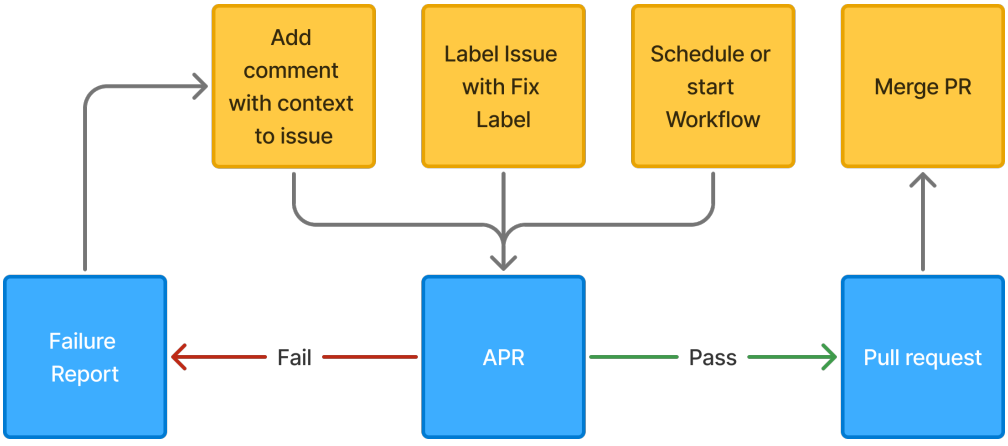

Figure 6.7.: *Resulting flow diagram*

sectionEvaluation Results

In this section, we present the results of the quantitative evaluation of the implemented APR prototype. This 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 3.2.

6.1.1. Validity

- GitHub runners introduce a lot of computational noise - Only a small set of runs were performed - results for average times do not fully ad up to apr core because the timing where stoped when an issue was completed not when the APR core finished

We asses more threats to validity in section 7.1.

6.1.2. Baseline of Evaluation

The resulting data is based on the executions of the APR prototype in the prepared repository (see 3.1.3), which contains all 40 issues from the QuixBugs benchmark. For evaluating the effectiveness, performance, and cost, we ran the APR prototype using eleven selected LLM models defined in 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.

6.1.3. 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. The first table 6.1 shows the results of the evaluation with one attempt per issue, while the second table 6.2 shows the results with the retry loop enabled and the max attempts set to 3.

Model	Repair Success Rate	Average Cost Per Issue	Average Execution Time in seconds
gemini-2.0-flash-lite	87.5%	$0.0001	11.33s
gemini-2.0-flash	87.5%	$0.0002	8.18s
gemini-2.5-flash-lite	90.0%	$0.0002	5.07s
gemini-2.5-flash	92.5%	$0.009	20.46s
gemini-2.5-pro	95.0%	$0.07130	70.09s
gpt-4.1-nano	70.0%	$0.0001	6.73s
gpt-4.1-mini	90.0%	$0.0007	8.96s
gpt-4.1	90.0%	$0.0033	7.42s
o4-mini	90.0%	$0.0069	23.08s
claude-3-5-haiku	0.0%	$0.0024	9.17s
claude-3-7-sonnet	87.5%	$0.0069	11.72s
claude-sonnet-4-0	90.0%	$0.0103	12.96s

Table 6.1.: *Zero shot evaluation results*

Model	Repair Success Rate	Average Cost	Average Execution Time
gemini-2.0-flash-lite	85%	$0.0003	13.08s
gemini-2.0-flash	90.0%	$0.0004	8.27s
gemini-2.5-flash-lite	90.0%	$0.0003	7.18s
gemini-2.5-flash	95.0%	$0.0111	23.65s
gemini-2.5-pro	97.5%	$0.07086	68.78s
gpt-4.1-nano	90.0%	$0.0003	10.61s
gpt-4.1-mini	97.5%	$0.0043	11.98s
gpt-4.1	97.5%	$0.004	7.45s
o4-mini	100%	$0.007	21.54s
claude-3-5-haiku	15.0%	$0.0076	23.98s
claude-3-7-sonnet	95.0%	$0.0085	27.16s
claude-sonnet-4-0	92.5%	$0.0117	16.80s

Table 6.2.: *Few shot evaluation results*

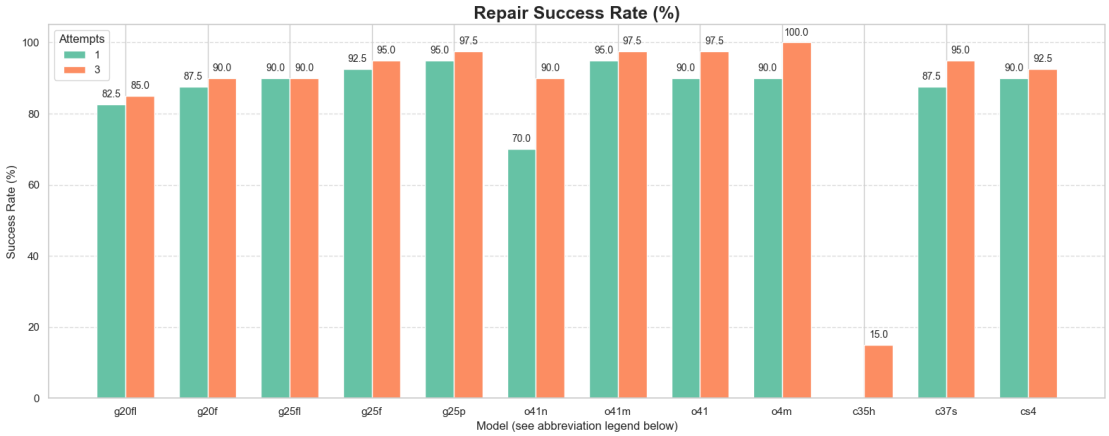

Figure 6.8.: *Repair Success Rate per Model*

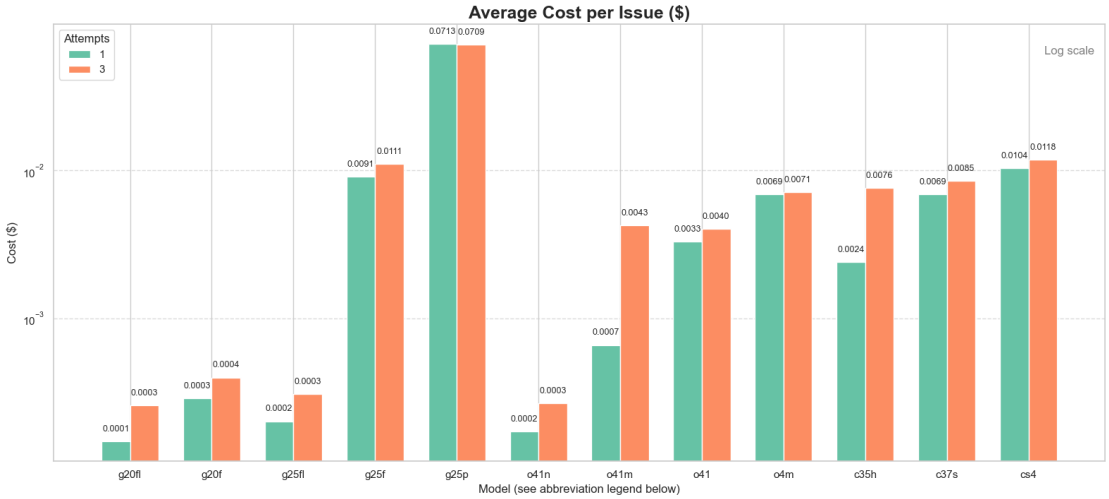

Figure 6.9.: *Average Cost per Issue per Model*

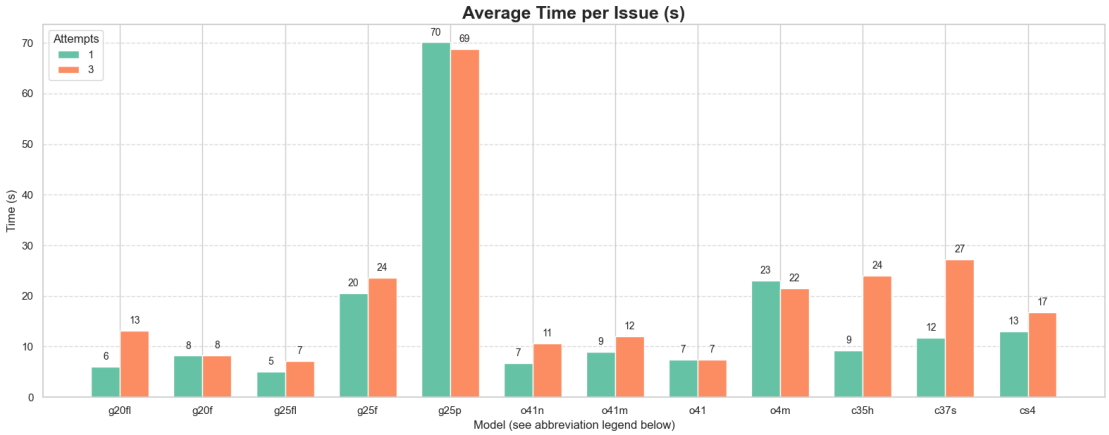

Figure 6.10.: *Average Execution Time per Issue per Model*

CI Overhead

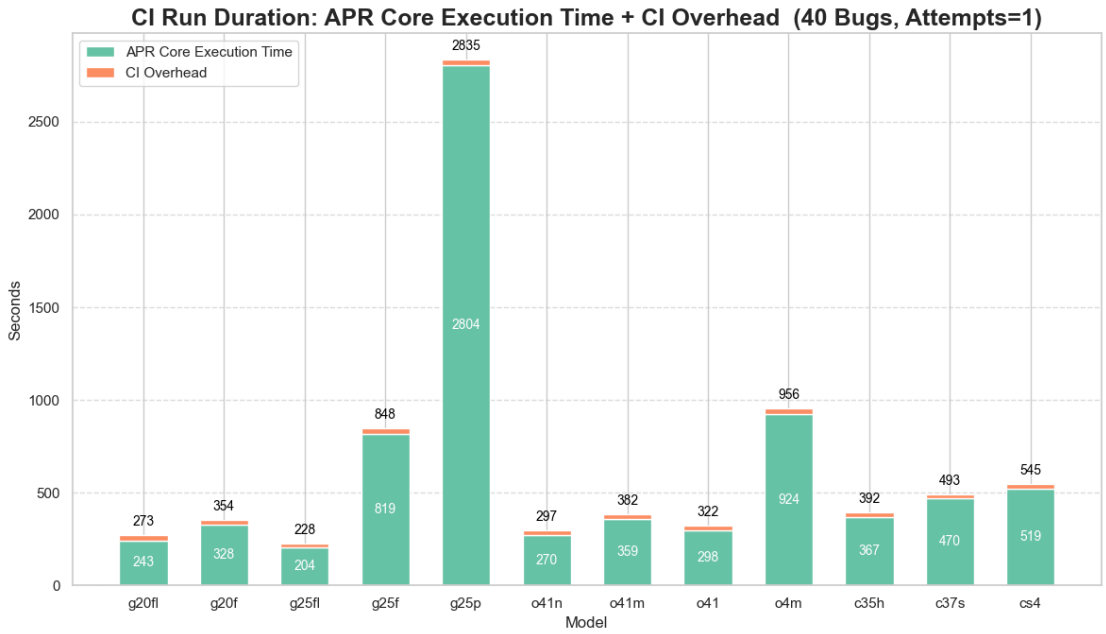

Figure 6.11.: *CI Overhead per Run with 1 Attempt*

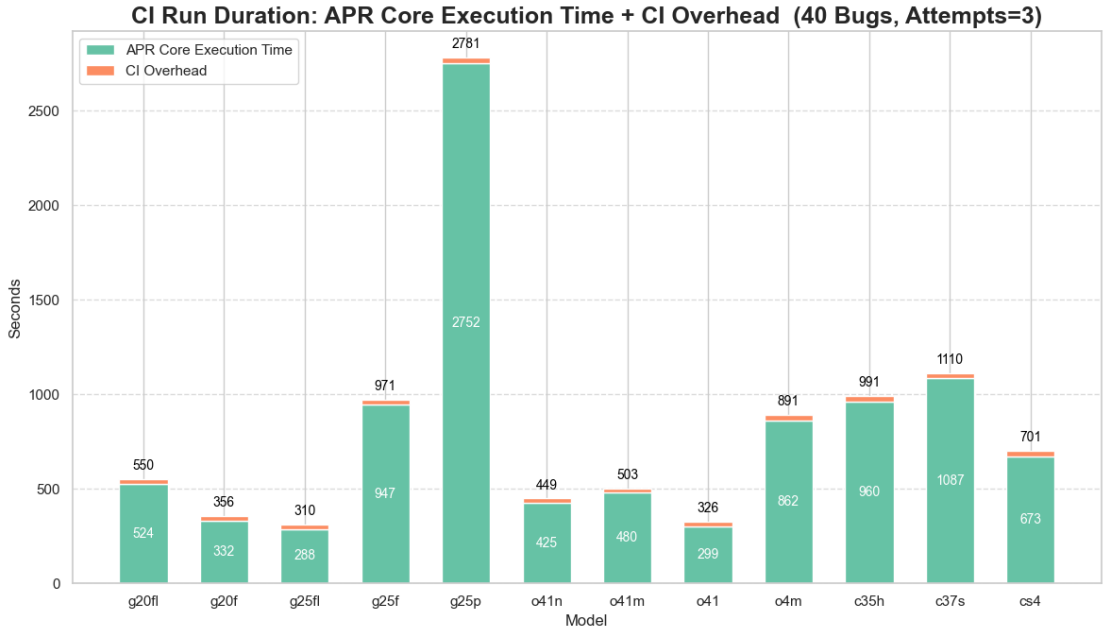

Figure 6.12.: *CI Overhead per Run with 3 Attempts*

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 evaluating the validity of our findings, the potential of our approach, and its limitations, and summarize the lessons learned. Finally, we outline a roadmap for future extensions of our work.

7.1. Validity

The results from the evaluation 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 once per model, these results are not fully representative of each model's performance. Furthermore, the speed and availability of the tested LLMs heavily depend on the providers' APIs, which can cause fluctuating execution times or even failures during periods of high traffic.

Costs are calculated based on token usage reported by the providers' API responses. However, these figures are not fully accurate, as providers are typically non-transparent about actual token counts [65]. 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 while leaving the program semantically incorrect.

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 its individual bugs may have appeared in the training data of some 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, architecture, and cost, which provides insight into how different configurations might generalize to larger datasets or other CI 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 on 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 and relatively low cost.

7.2. Potentials

Section 5 answers RQ1 by demonstrating how LLM based Automated Bug Fixing can be integrated into a CI workflow using GitHub Actions and containerized APR logic. A key advantage of this approach is, it allows for adjustments and further improvement without the need for major changes to the system. The concept is applicable to other Python repositories but was only tested on the "quixbugs-apr" repo and the "bugfix-ci" development repository. The option for custom configuration makes it adaptable to different repositories and environments.

Section 6.1 demonstrated that with this integration getting a automatically generated fix for a bug only requires a single user action. Creating and labeling an issue in the GitHub repository. This shows that the approach can take over and submit fixes, without the need for developer intervention. Creating issues/ticket for features, adjustments or bugs is a key part in agile frameworks for tracking tasks for in each iteration. The results indicate that this integration can support developers in an agile lifecycle by automating the design, development and test stages for a bug, leaving only requirement specification in form of issues and review of generated fixes to the developers. Furthermore issue descriptions for such bugs can be minimal, as the issues for evaluation contained no detailed information on the bug. This allows developers to focus on more complex tasks, while the system takes care of the simpler bugs. Our results align with the findings of [1], who states that LLMs can accelerate bug fixing and enhance software reliability and maintainability.

The evaluation results in section 6.1 show that the prototype can achieve a repair success rate of up to 100% on the QuixBugs benchmark with the best performing model "o4-mini". This indicates that the approach taken can be effective in fixing single file bugs in a CI environment. When taking a deeper look at the results, we observed that the repair success rate is highly dependent on the LLM model used.

With one attempt for every issue[1] the best performing models already achieve a repair success rate of 97.5% on the QuixBugs benchmark, while smaller models like gemini-2.5-flash-lite gpt-4.1-mini achieve a repair success rate of 95% and 90% respectively. This shows that with zero shot prompting smaller models can achieve good results, but larger models are more effective in fixing bugs.

[1] zero shot prompting

The potential of smaller models like: X,Y,Z lies in their performance and costs effectiveness. The results show that smaller models can achieve a repair success rate of 95% with a significantly lower cost and execution time compared to larger models. For example X solves x% with an average cost of X taking an average time of X per issue while Y repairs Y% successful with only $Y average per issue and 1/3 of the time per issue. This indicates that smaller models can be used to automate bug fixing in a CI environments, keeping costs low and performance resulting in quicker submissions of fixes.

The introduction of multi attempt fixes with an internal feedback loop, enhances the effectiveness of the integration significantly for every single model tested. By allowing the LLM to retry fix generation with additional context on failure, the repair success rates climb an average of X% per model. Particularly smaller models benefit from the few shot approach because the success rates rise to factor of X while costs and execution times rise linearly. This shows that small models can archive similar performance to larger model with lower costs and better performance.

With average repair times reaching from x-Y per issue, the approach is feasible for use in a CI environment. Furthermore these times do not impose significant waiting times for developers and indicate that bugs can be fixed quickly and efficiently while developers focus on more important and complex tasks. With costs reaching from $X to $Y per issue, this approach is also relatively cost effective and affordable.

–zero shot inexpensive for bigger models while for smaller it makes sense for multi attempt

Overall repair success rates, execution times and costs demonstrate that the approach is feasible and can be used to automate bug fixing in a CI environment. does not take significant time and furthermore it can be run concurrently

Leveraging paradigms of an agentless approach added with interactivity of the GitHub project platform we could achieve results that hold up with current APR research [16] showing similar repair success rates on the QuixBugs benchmark while providing an integrated approach with transparent cost and performance insights.

As mentioned before the performance is highly dependant on the underlying LLMs used. As companies like OpenAI, Google and Anthropic are constantly improving their models, the approach taken in this thesis can be extended to use future models as they become available. This means that the system can be improved over time without the need for major changes making the approach future proof and adaptable to new developments in the field of LLMs. The portability, modularity and extendability allows for future opportunities mentioned in 7.5.

7.3. Limitations

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

As mentioned in section 7.1, the system is constrained to addressing small issues only. Even with small issues, the timing and availability of the system are highly dependent 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. Runs cannot be skipped, and the filtering logic for events using external configuration data is very 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 do 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 that listens to webhook events and only triggers the workflow for relevant issues.

Additionally, security and privacy concerns arise from the fact that 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 to fix a bug 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. The prototype was not tested using open source models, but it is designed to be modular and extendable for use with open source models. Nevertheless, copyright and licensing issues may arise when code is generated by LLMs trained on copyrighted data [42, 1].

7.4. Lessons Learned

The development and evaluation of the prototype was an interesting and insightful experience. The following lessons were learned during the process: - LLMs can be used to automate bug fixing in a CI environment, but the results are highly dependent on the quality of the LLMs. - The apis of llm providers can be unreliable especially googles gemini- 2.5-pro. - The field of LLMs is rapidly evolving, and new models are released frequently. This makes it difficult to keep up with the latest developments.

7.5. Roadmap for Extensions

As mentioned before the prototype developed in this thesis is modular and easily extendable. This allows for future opportunities and research. Below we list potentials for further extensions and analysis which was not implemented because of the contained time for thesis.

- dig further into collected data, see where it went wrong in the patches or localization, further analyze the collected data, we collected a lot of data which can be used for more insights - test how cost, time and repair success can be optimized - small model with fallback to larger model - more complex prompt and context and prompt engineering - richer benchmarks - Service Accounts for transparent and better platform integration a - github app which replies on webhook events for better integration - test complex agent architectures and compare metrics and results - measure developer trust and satisfaction

8. Conclusion

References

[1] Xinyi Hou 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] Meghana Puvvadi 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] Emily Winter 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] Bogdan Vasilescu 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] Norbert Tihanyi 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] Feng Zhang 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] Cheryl Lee 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] Chunqiu Steven Xia 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] Yuwei Zhang 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] Jialin Wang and Zhihua Duan. *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] Yizhou Liu 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] John Yang 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] Dominik Sobania 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] Chunqiu Steven Xia and Lingming Zhang. "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] Haichuan Hu 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] Pat Rondon 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] Zhi Chen, Wei Ma, and Lingxiao Jiang. *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] Vincent Ugwueze and Joseph Chukwunweike. "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] Nayan B. Ruparelia. "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] Pekka Abrahamsson 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] Wael Zayat and Ozlem Senvar. "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] Ming Huo 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] Paolo Di Francesco, Ivano Malavolta, and Patricia Lago. "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).

[25] Alexey Tregubov 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).

[26] Michael Hilton 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).

[27] Taher Ahmed Ghaleb, Daniel Alencar Da Costa, and Ying Zou. "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).

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

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

[30] *About Issues*. 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).

[31] *About Pull Requests*. https://docs-internal.github.com/_next/data/LM3KAkw0Ii4E7Nz5N0zHV/en/
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&produ
requests&restPage=collaborating-with-pull-requests&restPage=proposing-changes-to-your-work-with-pull-requests&restPage=about-pull-requests. (Visited on 07/20/2025).

[32] *Understanding GitHub Actions*. https://docs-internal.github.com/_next/data/LM3KAkw0Ii4E7Nz5N
pro-team%40latest/actions/get-started/understanding-github-actions.json?versionId=free-pro-team%40latest&productId=actions&restPage=get-started&restPage=understanding-github-actions. (Visited on 07/20/2025).

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

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

[35] Yupeng Chang 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).

[36] Humza Naveed 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).

[37] Jared Kaplan 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).

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

[39] 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).

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

[41] Yi Liu 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).

[42] Jaakko Sauvola 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).

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

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

[45] Quanjun Zhang 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).

[46] Johannes Bader 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).

[47] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. "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).

[48] Xin Yin 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).

[49] Claire Le Goues 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).

[50] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. "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).

[51] Yiting Tang. "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).

[52] Thibaud Lutellier 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).

[53] Qihao Zhu 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).

[54] Chunqiu Steven Xia and Lingming Zhang. "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).

[55] Soneya Binta Hossain 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).

[56] Avinash Anand 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).

[57] Xiangxin Meng 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).

[58] Fairuz Nawer Meem, Justin Smith, and Brittany Johnson. "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).

[59] Kaixin Wang 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).

[60] Derrick Lin 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).

[61] René Just, Darioush Jalali, and Michael D. Ernst. "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).

[62] Claire Le Goues 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).

[63] Carlos E. Jimenez 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).

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

[65] Guoheng Sun 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).

A. Appendix

A.1. Source-Code

The source code of the thesis is available on GitHub split in two repositories: Evaluation repository with Prototype:
`https://github.com/justingebert/quixbugs-apr`
Prototype: `https://github.com/justingebert/bugfix-ci`

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.

_____

Datum, Ort, Unterschrift