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 ...

Contents

List of Figures

List of Tables

Listings

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 remarkable capabilities in understanding and generating code, LLMs have become valuable tools for developers' everyday tasks such as requirements engineering, code generation, refactoring, and program repair [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]. Software bugs have direct impact on software quality by causing crashes, vulnerabilities or even data loss [5]. The process of bug fixing can be time-consuming, leading to delays in software delivery and increased costs. 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 be effectively used to enhance automated bug fixing, thereby introducing new standards in the APR world showing potential of making significant improvements in 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 lifecycles and tooling 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. Modern software development makes use of continuous integration to ensure rapid, reliable releases. [19] By leveraging the capabilities of LLMs, we aim to develop a cost-effective prototype for automated bug fixing that seamlessly integrates using continuous integration (CI) pipelines. Considering computational demands, complexity of integration and practical constraints we aim to provide insights into possibilities and limitations of our approach 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 developer workflow enhancement?

The thesis is organized as follows:

Section 2 provides theoretical background on the Software Development, Generative AI in the context of software development and Automated Program Repair.
Section 4 and 5 go into the process of developing the prototype based on the requirements and methodology.
Section 6 showcases the resulting workflow and evaluation results for the Quixbugs benchmark.
Section 7 discusses the results and limitations of the prototype giving insights into lessons learned and a future outlook.
Finally section 8 concludes the thesis by summarizing the findings and contributions of this work.

2. Background and Related Work

In this section we present the essential theoretical background and context for this thesis. First introducing fundamental concepts in software engineering, the software development lifecycle (SDLC), continuous integration (CI), and the software project hosting platforms. The second part explores the rising role of GenAi/LLMs in software development practices.The third part showcases the evolution and state of APR and explores existing approaches.

2.1. Software Engineering

The following section introduces core concepts starting with the software development lifecycle,the importance of Continuous Integration (CI) in modern software development and the role of code hosting platforms.

2.1.1. Software Development Lifecycle

Engineering and developing software is complex process, consisting of multiple different tasks. For structuring this process software development lifecycle models have been introduced. These models evolve constantly to adapt to the changing needs of creating software. The most promising and widely used model today is the Agile Software Development Lifecycle [20].

The Agile lifecycle brings an iterative approach to development, focusing on collaboration, feedback and adaptivity. The Goal is frequent delivery of small functional features of software, allowing for continuous improvement and adaptation to changing requirements. Using frameworks like Scrum or Kanban, an Agile iteration can be applied in a development environment[20].

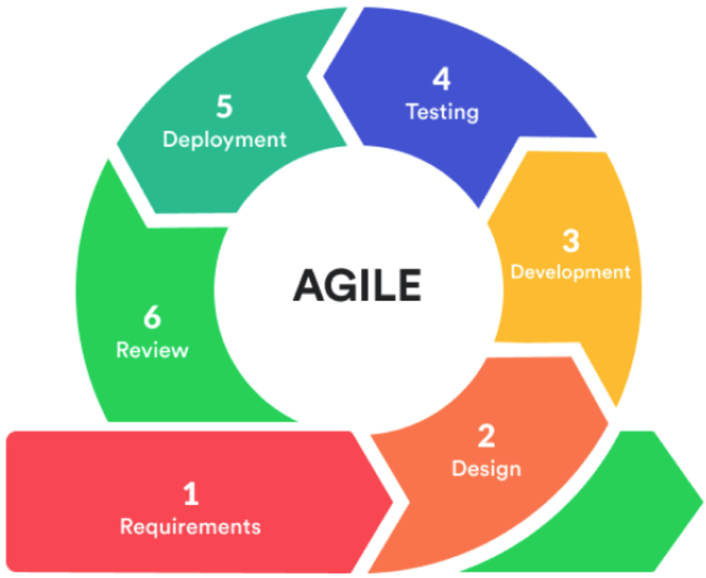

Figure 2.1.: *Agile software development lifecycle*

An Agile Software Development Lifecycle iteration consists of 6 key stages like in Figure 2.1 starting with planning phase where requirements for the iteration are gathered and prioritized. Secondly the design phase where the architecture and design of the feature is created. The third stage is where the actual development of the prioritized requirements takes place. After that the testing phase follows, where the software is tested for bugs and issues. The fifth stage is deployment, where the software is released to users. Finally, the changes are reviewed in a collaborative way.

When bugs arise during an iteration requirements can be reprioritized and the iteration can be adapted to fix these issues. This adaptivity is a key feature of Agile software development, allowing teams to respond quickly to changing requirements and issues but also slowing down delivery of planed features [20].

Modern software systems are moving towards lightly coupled microservice architectures, which results in more repositories which are smaller in scale tailored towards a specialized domain. This trend is driven by the need for flexibility, scalability, and faster development cycles. Smaller code repositories allow teams to work on specific components or services independently, reducing dependencies and enabling quicker iterations. This approach aligns with modern software development practices, such as microservices architecture and agile methodologies. With this trend developers work on multiple projects at the same time, which can lead to more interruptions and context switching when problems arise and priorities shift.

2.1.2. Continuous Integration

For accelerating the delivery of software in an iteration continuous integration has become a standard in agile software development. The main objective of continuous integration is to accelerate phases 3 and 4 [19]. CI allows for frequent code integration into a code repository. Automating steps like building and testing into the development resulting in rapid feedback right where the changes are committed in a shared repository. This supports critical aspects of agile software development, like fast delivery, fast feedback and enhanced collaboration [19].

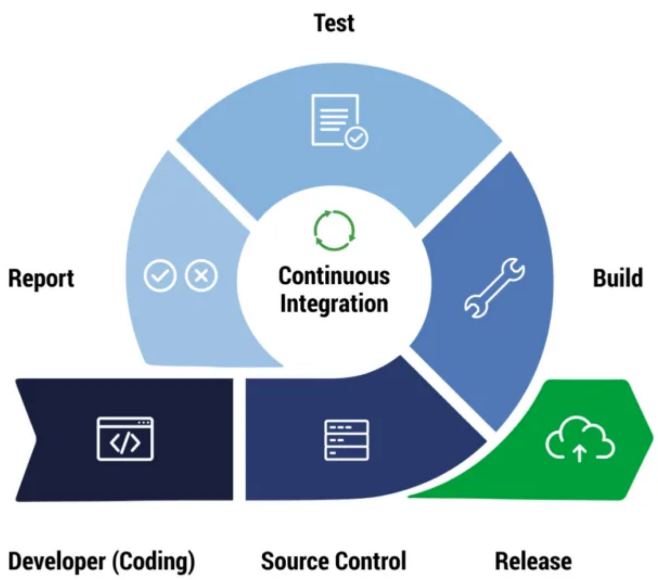

Figure 2.2.: *Continuous Integration cycle*

Although CI bring a lot of potential to agile development it can also has drawbacks. Long build durations and high maintenance.

2.1.3. Software Project Hosting Platforms

Software projects live on platforms like Github or GitLab. With GitHub being the most popular and most used for open source These 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. [21]

Github issues are a key feature of Github allowing for project scoped tracking of features, bugs, and tasks. 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.

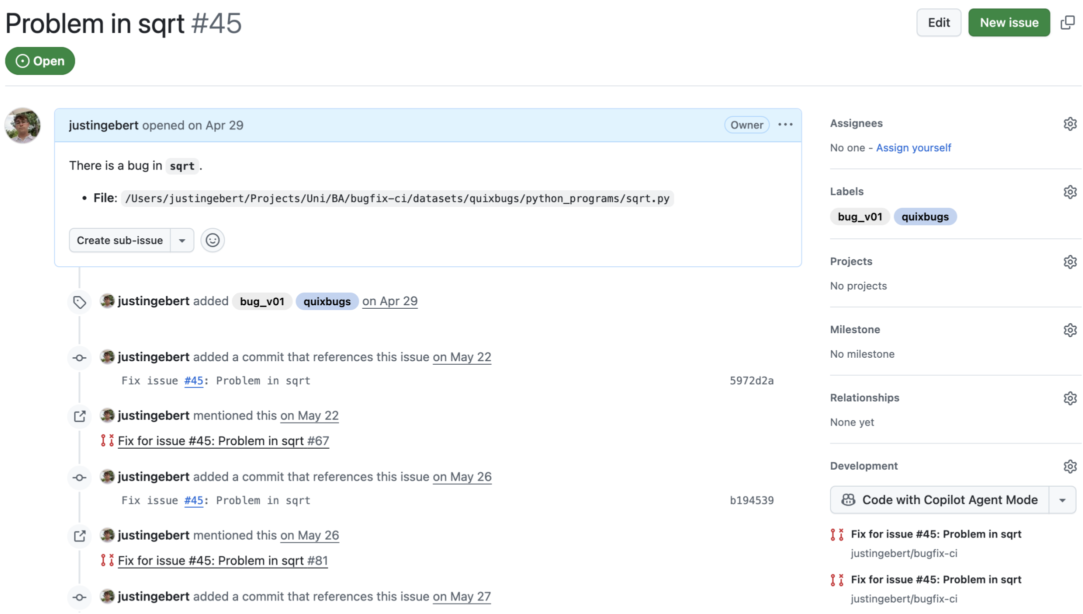

Figure 2.3.: *Example of a GitHub Issue*

For integrating and reviewing code into production GitHub provides Pull Requests. A Pull Request proposes changes to the codebase, providing an integrated review process to validate changes before they are integrated into the production codebase. Code changes are displayed in a diff format [1] allowing reviewers to see and dig into the changes made.This process is essential for maintaining code quality and ensuring that changes are validated before being merged. Pull requests can be linked to Issues, allowing for easy tracking of changes related to specific tasks or bugs.

GitHub also provides a manged solution (Github Actions) for integrating CI into a repositories by writing CI workflows in YAML files. Workflows can run as CI pipelines on runners hosted by GitHub or self hosted runners. A workflow consists of triggers and jobs, and steps. One or more events can trigger a workflow which executed one ore more jobs which are made up of one or more steps. [22] An example is shown in Figure 2.4. Workflow results and logs can be viewed from multiple points in the GitHub web 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.

[1] TODO explain format

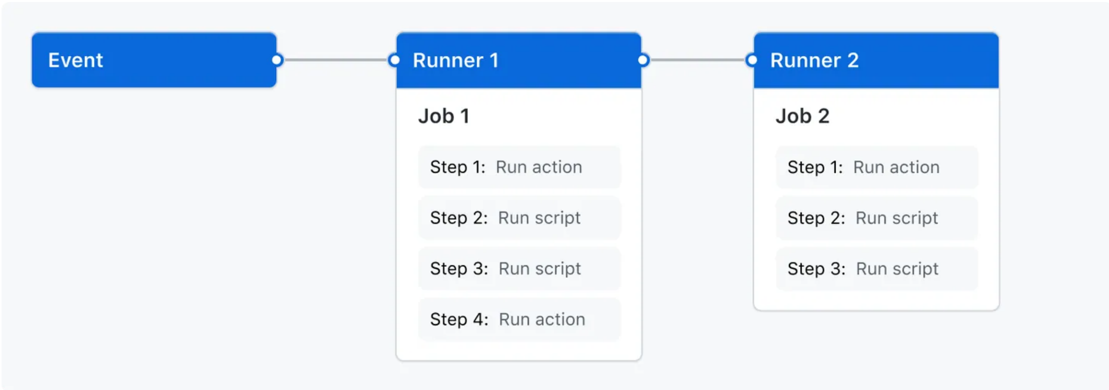

Figure 2.4.: *Building blocks of a Github Action*

2.2. Generative AI in Software Development

This section will cover the role of Generative AI in software development. First we will define Generative AI and Large Language Models (LLMs). The second part will focus on the impact of Generative AI on software development practices.

2.2.1. Generative AI and Large Language Models

Generative Artificial Intelligence (Gen AI) is subfield of artificial intelligence and refers to systems that can generate new content based on patterns learned from massive amounts of training data. Advanced machine learning techniques, particularly deep learning, enable these systems to generate text, images, or code, that resembles human-generated output. In the field of natural language processing (NLP) was revolutionized by the Transformer architecture [23]. It lays the ground work for Large Language Models which are specialized in text generation. Extensive training data results in Models billions of parameters, allows them to understand and generate human-like text in multiple natural languages and diverse programming languages. However this requires enormous computational resources during training and operation [24]. A models parameter count has a direct impact on the model's performance, with larger models generally achieving better results in various NLP tasks but also demanding more computational resources Despite modern LLMs showing promising results in text generation, they can still hallucinate incorrect or biased content. [24].

To archive a specific task using LLMs designing and providing specific input to the model to guide its output is called prompt engineering. This process is crucial for achieving desired results from LLMs, as the quality and specificity of the prompt directly influences the model's output. The input is constrained by a models context window, which is the maximum amount of text the model can process at once.

Popular Large Language Models are offered via APIs but providers like OpenAi, Anthropic and Google, or open source alternatives like X. A comparison of popular LLMs is shown in Table 3.1.

2.2.2. Large Language Models in Software Development

Large Language Models are reshaping software development by automating various tasks. They have billions of parameters and are pre-trained on massive codebases which results in extraordinary capabilities in this area [18]. Tools like Github Copilot, OpenAI Codex, and ChatGPT have become popular in the software development community, providing developers with AI-powered code suggestions and completions [25]. These tools get applied in various stages of the software development lifecycle, including requirement engineering, code generation, debugging, refactoring, and testing [1, 2, 25]. By using LLMs to enhance the named tasks development cycle times can be reduced by up to 30 percent [25, 26]. Furthermore these tools have positive impacts like improving developer satisfaction and reducing cognitive load [26].

Although Generative AI gets adopted really quickly in many areas of software development this technology still faces limitations. LLMs have challenges working on tasks that are outside their scope of training or require specific domain knowledge [1]. Additionally LLMs have limited context windows, which can lead to challenges when working with large codebases or complex projects where context windows are too small for true contextual or requirements understanding [25]. When generating code LLMs can produce incorrect or insecure code, which can lead to further bugs and vulnerabilities in the software [1, 25]. Additionally when integrating LLMs into tools can be vulnerable to prompt injection, where unintended instructions are injected at some point and can also lead to production of harmful code [27]. Code generated by LLMs based on training data also raises questions about ownership, responsibility and intellectual property rights. [28, 1].

Facing these challenges, different approaches have been developed. AI Agents, RAG or interactive approaches are prominent examples. These approaches aim to enhance the capabilities of LLMs by providing additional context, enabling multi-step reasoning, or allowing for interactive feedback loops during code generation and debugging [1, 2]. Section 2.3.1 will go into more detail on these approaches.

Recently research is exploring solutions which integration LLMs into existing software development practices and workflows. [2, 29, 30, 28]. This happens in tools and on platform where development happens and focuses on for example integrating AI/ML into CI/CD [31] or into code hosting platforms like GitHub 2.1.3.

2.3. Automated Program Repair

Automated Program Repair (APR) is used to detect and repair bugs in code with minimal human intervention. [32] APR systems are supposed to take over the process of fixing bugs, reducing load for developers and making time to focus on more relevant work. [1]

Using APR systems specific bugs can be fixed using a generated patch. Working patches are usually generated using a 3 stage approach: First localizing the bug. Then repairing the bug, in the end validation decides where the bug will be passed on [32, 33]. This approach is similar to the bug fixing process of a developer, where the bug is first identified, then fixed, and finally tested and reviewed to ensure the fix works as intended. An example of an APR system being applied at scale is Getafix, which is used at Meta to automatically fix common bugs in their production codebase [33].

The field of Automated program repair also greatly benefited of the rapid advancements in AI. With new research and benchmarks setting new standards in the field.[2, 1]

In this section we will provide an overview of the evolution of APR, related work, and the current state of APR systems. Followed by a display of the most common APR benchmarks used in research.

2.3.1. Evolution of Automated Program Repair

We have seen multiple paradigm shifts in the field of Automated Program Repair (APR) over the years. This evolution of APR can be categorized into key stages, each marked by significant advancements in techniques and methodologies.

Traditional Approaches:

Traditional APR approaches typically rely on manual crafted rules and predefined pattern. [12, 34, 35]. These methods are generally classified into three main categories: search based, constraint/semantic based, and template-based repair techniques.

- **Search based repair** searches for the correct predefined patch in 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 based on fitness determined by test cases [36].
- **Semantics / constraint based repair** synthesizes patches using constraint solvers to based on semantic information of the program and tests. [12, 37] Angelix being a prominent example. [37].
- **Template based repair** relies on mined templates for transformations of known bugs. [34] Templates are mined from previous human produced bug fixes.[34, 35]. Getafix being an example of an industrially deployed tool learning recurring fix pattern form past fixes [33]

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

Learning based Approaches:

Learning based APR introduced machine learning techniques to the field, improving the number and variety of bugs that can be fixed. Deep neural networks using bug fixing patterns from historical fixes as training data, learn how to generate patches to "translate" buggy code into correct code [34, 38]. A prominent of a learning based APR system is CoCoNut [39], Recoder [40]. Despite significant advancements these methods are limited by training data and struggle with unseen bugs. [41]

The emerge of LLM based APR:

The explosive growth of LLMs has transformed the APR space. LLM based APR techniques have demonstrated significant improvements over all other state of the art techniques, benefitting from the coding knowledge [42]. For that reason LLMs lay the groundwork of a new APR paradigm [18, 43].

Different approaches leveraging LLMs have emerged and are being actively researched. These include:

- **Retrieval-Augmented approaches** repair bugs with the help of retrieving relevant context during the repair process. For example code documentation stored in a vector database [2]. This approach allows access to external knowledge in the repair process, enhancing the LLM's ability to understand and fix bugs [1, 35].
- **Interactive/Conversational approaches** make use of LLMs dialogue capabilities to provide patch validation with instant feedback. [15, 16] This feedback is used to iterate and refine generated patches with the goal of archive better results. [15]
- **Agent based system** improve bug localization and fixing by equipping LLMs with the ability to access external environments, operate tools (like file editors, terminals, web search engines), and make autonomous decisions. [43, 2, 44] Using multi-step reasoning these frameworks reconstruct the cognitive processes of developers using multiple specialized agents. [17, 10, 8]. Examples include SWE-Agent [13], FixAgent [8], MarsCodeAgent [12], GitHub Copilot.
- **Agentless systems** are a recent push towards more lightweight solutions, focusing on simplicity and efficiency. These approaches aim to reduce the complexity of APR systems by cutting complex multi-agent coordination and decision making, while maintaining effectiveness in bug fixing [9, 2]. This approach provides clear rails to the LLMs improving transparency of the bug fixing approach. Using 3 steps localization, repair, validation promising results with low costs have been achieved using this paradigm [9, 44].

Commonly used LLMs for the mentioned APR techniques include ChatGPT, Codex, CodeLlama, DeepSeek-Coder, and CodeT5 [1, 35, 43]. Despite significant advancement state of the art APR system still face challenges and limitations. Existing system suffer from complexity with limited transparency and control over the bug fixing process.[9, 2, 1] The bug fixing process bugs takes a lot of computational resources and is time intensive making the program repair expensive [14, 2]. APR system are build and applied in controlled environments making APR unreachable for developers since they cant be integrated into real world software development workflow and projects[45, 2]

2.3.2. APR benchmarks

For standardizing evaluation in research of new APR approaches benchmarks have been developed. These benchmarks consist of a set of software bugs and issues, along with their corresponding fixes or tests, which can be used to evaluate the effectiveness of different APR techniques. [43] They are essential for comparing the performance of different APR systems and understanding their strengths and weaknesses. [2] APR benchmarks are available for different programming languages with popular ones being QuixBugs [46], Defects4J [47], ManyBugs [48] and SWE Bench [49]. [50]

Model	**Languages**	**Number of Bugs**	**Description**	**Difficulty**
QuixBugs	Python, Java	40	small single line bugs	Easy
Defects4J	Java	854	real-world Java bugs	Medium
ManyBugs	C	185	real-world C bugs	Medium
SWE Bench	Python	2294	Real GitHub repository defects	Hard
SWE Bench Lite	Python	300	selected real GitHub defects	Hard

Table 2.1.: *Overview of APR benchmarks*

3. Method

The primary objective of this thesis is to assess the potentials and limitations of a self developed APR pipeline prototype. 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 key potentials of this integrated approach in terms of repair success rate, cost-effectiveness and developer workflow enhancement?
- **RQ3:** What are the primary limitations and challenges, such as performance overhead, accuracy, and security, of using LLM-based APR within a CI context?

For answering these questions we streamlined this process into three phases Preparation, Implementation and Evaluation, shown in Figure 3.1.

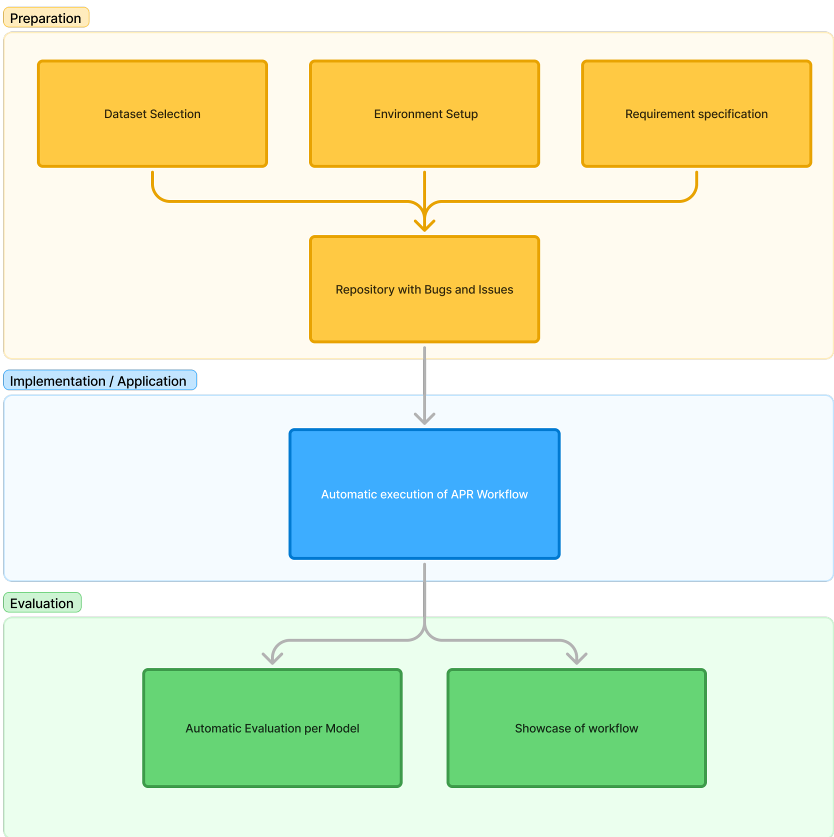

Figure 3.1.: *Thesis methodology approach*

In the preparation we select a suitable APR benchmark. With this 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 evaluation of the self developed prototype is done by using the defined evaluation metrics 3.3 collected during the execution of the APR system. Furthermore we will showcase the resulting workflow of using the system in a repository. The following sections will go into detail of 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 and used. This includes selecting a suitable dataset, setting up the environment, and specifying the requirements for the system.

3.1.1. Dataset Selection

For the evaluation of our APR integration, we selected the QuixBugs benchmark [46]. This dataset is well-suited for our purposes due to its focus on small-scale software bugs in Python. It consists of 40 individual files containing an algorithmic bug each. The bug is always caused by single erroneous line. QuixBugs brings corresponding tests and a corrected version for every file which allows for repair validation. The bugs where developed as challenging problems for developers [46], it enables us to evaluate if our system can take over the cognitive demanding task of fixing small bugs without developer intervention.

Compared to other APR benchmarks 2.1 like SWE-Bench [49] 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 several LLM models. multiple smaller models for this benchmark to. these models are to have fast response times and low costs. For that we will compare a wide selection of recent (point 11.07.2025) models from different providers. The models are selected to cover a range of capabilities and costs in the lower tier, allowing us to evaluate the performance and cost-effectiveness of the APR system. The selected models are:

3.1.3. Environment Setup

To mirror realistic software development environment, we prepared a GitHub repository containing the QuixBugs datasets python files. This repository serves as the basis for the bug fixing process, allowing the system to interact with the codebase and perform repairs. The repository contains only relevant files and folders required for the bug fixing process, ensuring a clean environment for the system to operate in.

Model Name	Publisher	Context Window Size in Tokens	Cost per 1M Tokens	Benchmark performance
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
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-haiku	Anthropic	200,000	input: $0.25 output: $1.25	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

Table 3.1.: *Characteristics of selected Large Language Models*

Using the relevant files we generate a GitHub issue for each bug, using a consistent template that captures only the title of the Problem. These issues serve as the entry points and communication medium to our APR pipeline.

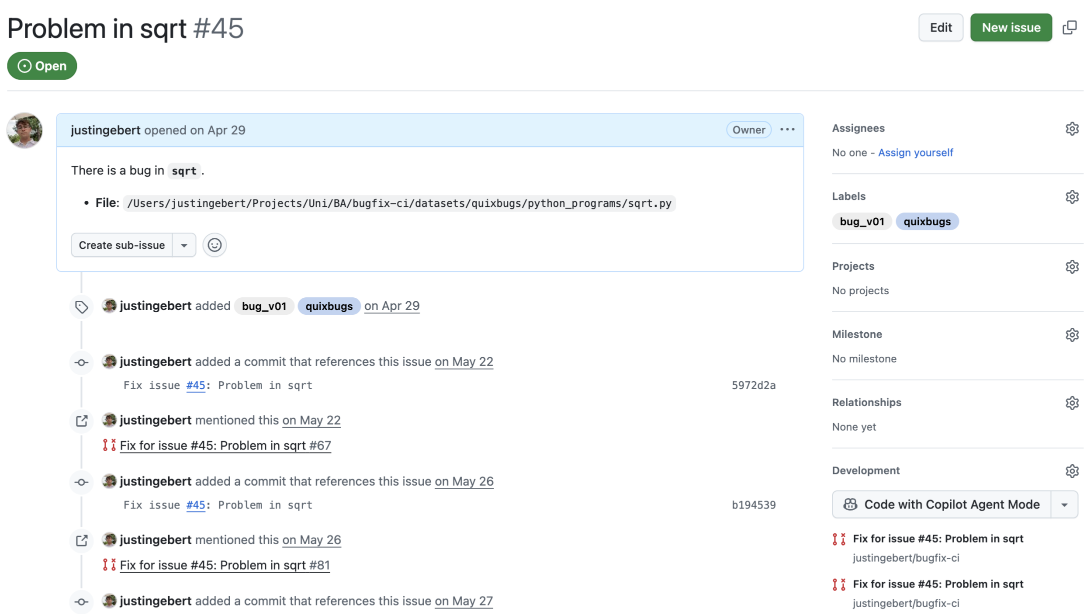

Figure 3.2.: *Example of a GitHub Issue*

3.1.4. Requirements Specification

Before implementation we constructed the requirements for the prototype following the INVEST model, a widely-adopted method in Agile software development for engineering requirements [51]. According to the INVEST principles, each requirement was formulated to be independent, negotiable, valuable, estimable, small, and testable. This model ensured that both functional and non-functional requirements were precisely defined, clearly verifiable, and easily adaptable to iterative development. The resulting requirements are detailed in 4.

3.2. Pipeline Implementation

In this section we will give a high-level overview of the implemented Automated Bug Fixing Pipeline. More detailed information about the implementation can be found in 5.

The Automated Bug Fixing Pipeline was developed using iterative prototyping and testing, with a focus on simplicity and extendability. Using the self developed requirements4 we build the following System:

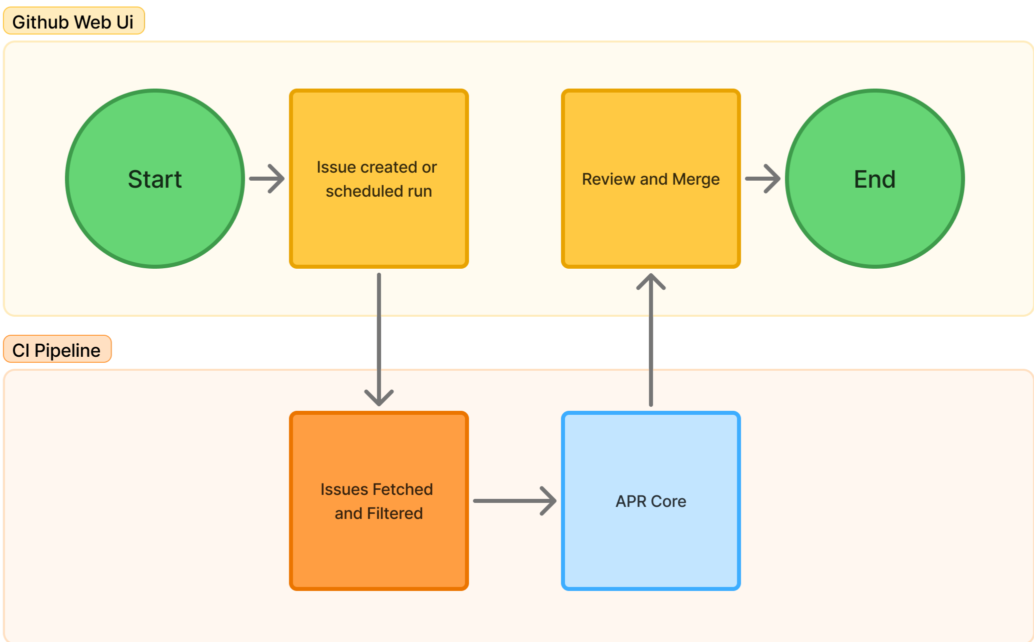

Figure 3.3.: *High-Level overview of the APR Pipeline*

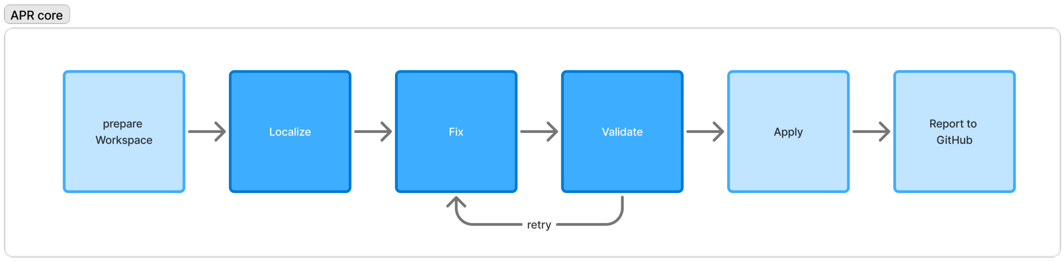

Figure 3.4.: *High-Level overview of the APR Core*

When the system is in place in a target repository the pipeline can automatically be triggered by the configured triggers. On successful trigger a github action runner executes the pipeline. After entrypoint the relevant issues are selected and filtered for processing. The issues are the passed to the Dockerized APR system which talks to the configured LLM via API to localize and fix the issue. With the generated file edits the changes is validated and tested. When validation passes changes get applied and a pull request is automatically opened on the repository, linking the issue and providing details about the repair process. In case of a unsuccessful repair or exhaustion of the configured maximum attempts the failure is reported to the issue.

3.3. Evaluation

In this section, we describe how we measure the effectiveness, performance and cost of our APR pipeline when integrated into a Github repository. Using Github Actions Continuous Integration abilities, with our QuixBugs repository as a bases. For this evaluation we selected a set of LLM models to be used for the APR process.

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

For evaluating the effectiveness of the LLM model used and approach the determine the repair success rate. A success of an issue is determined by validation and the complete test suite provided by QuixBugs. An issues is considered successfully repaired when the validation passes with correct syntax and the test suite passes all tests.

Furthermore we will evaluate wether multiple attempts to repair an issue improve the repair success rate and how the number of attempts relates to the cost of repairing an issue.

For evaluating feasibility and performance evaluation we will analyze collected aggregated and fine grained timings and costs of a repair attempt.

The following data is automatically collected by each run of the APR pipeline. Using self developed scripts we collect the data from different sources for each run. Using this data to calculate the results defined in 3.5

Metric	Description	Source
Run ID	Unique identifier for each run of a Workflow	Github Action Env
Model Used	LLM model used for the repair process	Configuration File
Configuration	Configuration details, including LLM Model and settings	Target Github repository
Successful Repairs	Count of issues successfully repaired	APR Core
Execution Time	Total time taken for the run	APR Core
Job Execution Times	Time taken for each job in the pipeline	Github API
Issues Processed	Information of Issues processed in the run 3.3	APR Core

Table 3.2.: *Run metrics collected for each execution*

Data collected for each issue processed by the APR core:

Metric	Description	Source
Issue ID	Unique identifier of the issue processed	Github Issue ID
Repair Successful	Boolean indicating whether the repair was successful. Tr	APR Core
Number of Attempts	Total attempts made	APR Core
Execution Time	Total time taken to process the issue, including all stages	APR Core
Tokens Used	Number of tokens consumed by the LLM during the repair process	LLM API
Cost	Cost associated with the repair process, calculated based on tokens and execution time	APR Core
Stage Information	Details about each stage of the repair process, including execution time and outcome	APR Core

Table 3.3.: *Metrics collected for each processed issue*

An attempt to repair an issue consists of multiple stages during the repair process, each with its own metrics. The following table summarizes the metrics collected for each stage:

Metrics	Description	Source
Stage ID	Unique identifier for each stage in the repair process	APR Core
Stage Execution Time	Time taken for each stage of the repair process	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.: *Metrics collected for each stage of an issue repair attempt*

Using this data we calculate the following metrics for each run of the system. This data will be calculated for each run using a different LLM model from the set of selected models 3.1.2. The metrics will be used to answer the research questions and evaluate the performance of the APR system. The following table summarizes the metrics and their calculations:

Metrics	Calculation
Repair Success Rate	Number of Successful Repairs / Total Issues Processed
Average Number of Attempts	Total Attempts / Total Issues Processed
Average Execution Time	Total Execution Time / Total Issues Processed
Average Tokens Used	Total Tokens Used / Total Issues Processed
Average Cost	Total Cost / Total Issues Processed
Average Stage Execution Time	Total Stage Execution Time / Total Stages Processed
Average Stage Outcome Success Rate	Number of Successful Stages / Total Stages Processed

Table 3.5.: *Evaluation metrics calculated from the collected data for a run*

4. Requirements

For development we developed the following requirements which guide the implementation and design of the prototype. Following the INVEST model [51] we constructed functional 4.1 and nonfunctional 4.2 requirements.

These requirements allow for better planning and prioritization during implementation and tracked progress.

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, workspace/ contains the correct 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.	Logs/Context shows build and test results.
F6	Iterative Patch Generation (retry logic)	If F5 reports failures, retry F4-F5 up to `max_attempts` times.	After retries, either F4 passes or fails with no further retries.

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	Open a PR or post a comment on GitHub with the diff.	A PR or appears for each issue, showing diff and summary.
F9	Logs and Metrics Collection	Provide log files and Metrics with fix-rate, attempt history, timings, token usage.	A metrics file contains fields: issue, success, timings, stages.

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 via YAML.	Changing the config file alters agent behavior accordingly.
N3	Portability	The system can be deployable on any repository on GitHub.	???
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 after each run.

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 code is attached in the A appendix.

The goal was to create a system which not only fixes bugs but is also portable /deployable across different repositories and configurable to some extend.

The resulting system consists of two main components. The **APR core** which hold the core logic for the repair process and lives inside a docker image, making it portable and easy to deploy. 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 issues and configured triggers in the repository.

5.1. System Components

APR Core:

The APR core contains the main bug fixing logic, its written in python. Embedding it into a Docker Image [1] it remains easily portable and small in memory footprint. In order to use the APR core the following environment needs to passed to the container:

Table 5.1.: *Container Inputs*

Name	Description	Type
git repository	files where APR should look for fixes	docker volume mount
GITHUB_TOKEN	GitHub token for authentication and API access	environment variable
LLM_API_KEY	API key for the LLM provider to generate fixes	environment variable
ISSUE_TO_PROCESS	The issue to process, can be a single issue or a list of issues	environment variable

[1]link to docker

GITHUB_REPO	The GitHub repository where the issues are located	environment variable

With this environment set the APR core iterates over all issues which are fetched from the (ISSUE TO PROCESS) environment variable. For each issue the main APR logic is executed. This main logic is a predefined flow which is made up of the implemented stages and tools.

At first the workspace (branch checkout) and the issue repair context is set up. The context is the main data structure for the issue repair and is processed at every step.

```
context = {
        "bug": issue,
        "cfg": cfg,
        "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"),
            "issue_number": issue["number"],
            "issue_title": issue["title"],
            "execution_repair_stages": {},
            "repair_successful": False,
            "attempts": 1,
        },
    }
```

Listing 5.1: *Context JSON*

A stage uses the context to perform a specific task in the bug fixing process and returns the context with its added context. The implemented stages are Localize, Fix, Build and Test. The repair process starts with the localization stage. This stage localizes the files needed to fix the bug in the codebase using the configured LLM Model via the providers SDK/API. The prompt is build using the issue and a constructed hierarchy of the repositories file structure. The response is expected to return a list of files where the bug might be located.

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

```

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

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

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

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

Listing 5.2: *Localization Prompt*

With the localized files in the context the Fix stages comes next. Again this stage makes use of the configured LLM API to fix the localized files. A prompt contains the issue and file names with file content. The response is expected to contain a list of edits for each file while also allowing the LLM to specify that no changes need to be made in a file. The generated edits are then applied to the files in the workspace. The context is updated with the new file content.

```
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 validation 2 stages are available Build and Test. The Build stage is responsible for validating syntactics of the changes made in the Fix stage. It checks if the code can be built successfully. For python this means checking if the syntax is correct and follows standardized code quality rules [2]. This is archived by first formatting the code using the Python formatter Black[3] followed by linting using flake8[4].

If a test command is configured the Test stage is executed next. This stage runs the tests defined in the repository using the configured test command.

In case Build or Test fail, the context is updated with the error messages, and the system will retry the Fix stage with a new attempt. For attempts additional feedback is generated using the previous code and stage results with details.

When maximum number of attempts is reached and the code does not pass validation unsuccessful repair is reported to the issue by creating a comment using the Github API.

On successful Build and Test the issue is marked as a successfully repaired. The file changes are committed and a diff file is generated. The branch is pushed to the remote repository, and a pull request is created. The pull request contains the changes made in the Fix stage, and the issue is linked to the pull request.

During execution the APR core logs its actions, which can be used for debugging and provides transparency about the repair process. Furthermore it collects metrics such as the number of attempts, execution times, and token usage, which are essential for analyzing the effectiveness and performance of the APR system.

The agent 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/CD environment.

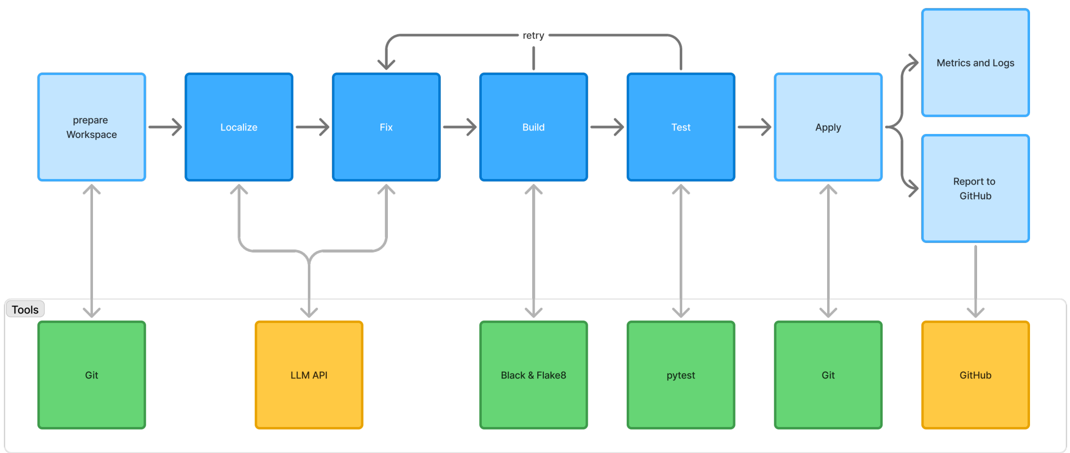

Figure 5.1.: *APR Core Logic*

Continuous Integration Pipeline:

[2] todo
[3] todo
[4] todo

The Github Action Workflow integrates the APR core into the desired GitHub repository. It is written in YAML[5] according to the Github Action standard. We will use runners provided and hosted by Github which takes away the overhead of managing our own runners but comes at the cost of unknown performance and availability. The Workflow is made up of multiple triggers and jobs. Triggers work by using events happening in the GitHub repository and serve as the entry point for executing the jobs. Given the triggers the workflow can is executed two different ways:

- Process all issues with desired state for repair. Triggered by manual dispatch ("workflow_dispatch") or scheduled execution ("cron").
- Process a single issue. Issue is labeled with the configured labels ("issue_labeld") or when extra information is added or edited on an issue in form of a comment ("issue_comment").

The trigger event information gets passed as environment variables to the first job. "gate" uses this data to evaluating if the issue should be processed or skipped using a python script (filter_issues.py). This script checks the labels and resolves the issue state to determine if the issue is relevant for the APR process. If no issues pass this "gate" the job "skipped" is executed, which simply logs that no issues were found to process and exits the workflow. When issues pass the "gate" the "bugfix" job is started. This job is responsible for executing the APR core logic. It provides necessary prerequisites for the APR core Docker container to run 5.1 and perform the repair. This includes checking out and mounting the repository, setting up environment variables, and providing the necessary permissions for the core to edit repository content, create pull requests, and write issues.

For giving access to the agent cores logs and metrics the job provides the logs directory as an artifact which is available after the workflow run is completed.

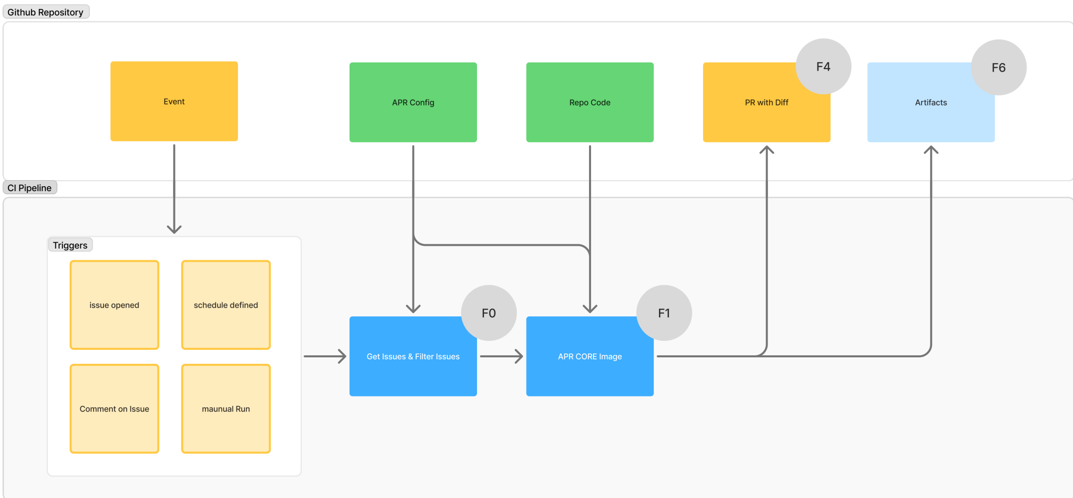

Figure 5.2.: *APR Core Logic*

[5]todo

To use this workflow in a repository, it needs to be placed in the '.github/workflows' directory of the repository along with the 'filter_issues.py' in '.github/scripts'. The following adjustments need to be made to the repository:

- Add Secret: LLM provider API key
- enable GitHub actions permission to create pull requests in repo settings -> actions -> General -> Workflow permissions (12.07.2025)

5.2. System Configuration

To fulfill requirement X some of the systems behavior can be controlled by adjusting a configuration. A configuration is optional when no configuration is in place the system will use default configuration which is defined in the code. The configuration is written in YAML and can be placed at the root of the repository. This allows for easy customization of the system without changing the code itself. The configuration file is named 'bugfix.yml' and is read by the APR core and CI Pipeline during execution. The configuration allows for controlling labels, workdir, branches, attempts and LLM models used for the repair process.

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 resides, used for mounting in the Docker container.
test_cmd	The command used to run tests on the codebase.
branch_prefix	The prefix for branches created for bug fixes, allowing for easy identification of bug fix branches.
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 before giving up.
provider	The LLM provider used for generating fixes.
model	The specific model from the LLM provider used for generating fixes.

The full implementation is listed in Appendix A

5.3. Requirement Validation

The following sections outline how each requirement was satisfied.

6. Results

In the following section we will present our results from the implementation and evaluation. For accessing the potentials and limitations of integrating LLM based Automated Bug Fixing into software development workflows using Continuous Integration we implemented a working prototype. During the implementation 5 all self developed requirements 4 are satisfied.

The application of the prototype on a real Github repository 3.3 is demonstrated in the first part of this chapter by showcasing the resulting workflow on GitHub. In the second part of this section we present the results of the quantitative evaluation the prototype being used on the prepared mentioned repository containing the QuixBugs benchmark.

6.1. Showcase of workflow

Setting up the APR system in a repository is archived by adding 2 required files and one optional configuration file to the repository. As seen in 6.1 the required files are the **.github/workflows/auto-fix.yml** file and the **.github/scripts/filter_issues.py** file. The optional configuration file is the **.bugfix.yml** file.

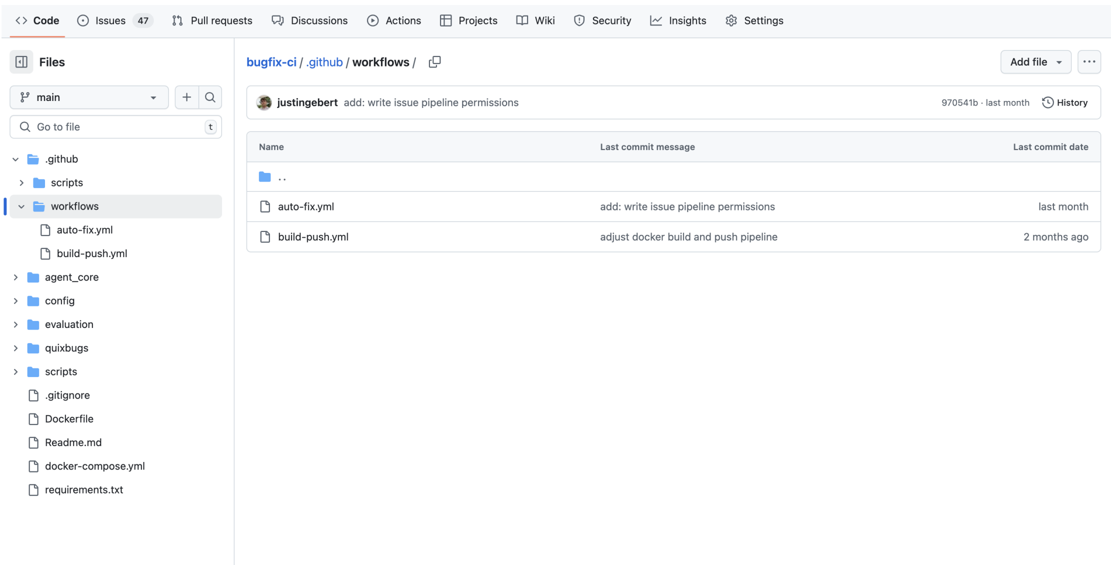

Figure 6.1.: *Set up of APR in a repository*

With the system in place and a custom configuration file set up, the APR system is ready to be used in the repository. Automated bug fixing can be applied in two ways. Processing one issue per run by labeling the issue with the with the default (bug_v01) or configured label will trigger the workflow and process the issue. This allows for fast feedback and quick bug fixing at issue creation

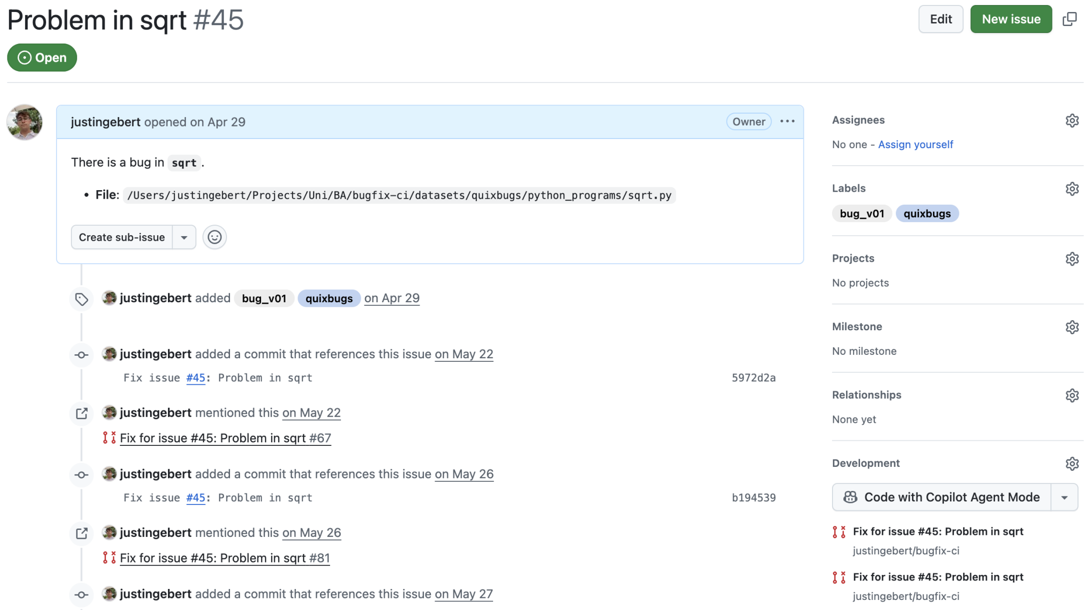

Figure 6.2.: *Trigger automatic fixing for single issue*

The second way is to process all issues labeled with the default (bug_v01) or configured label by scheduling the workflow to run at a specific time or dispatching it manually. This allows for a more automated approach to bug fixing, where issues are processed in batches.

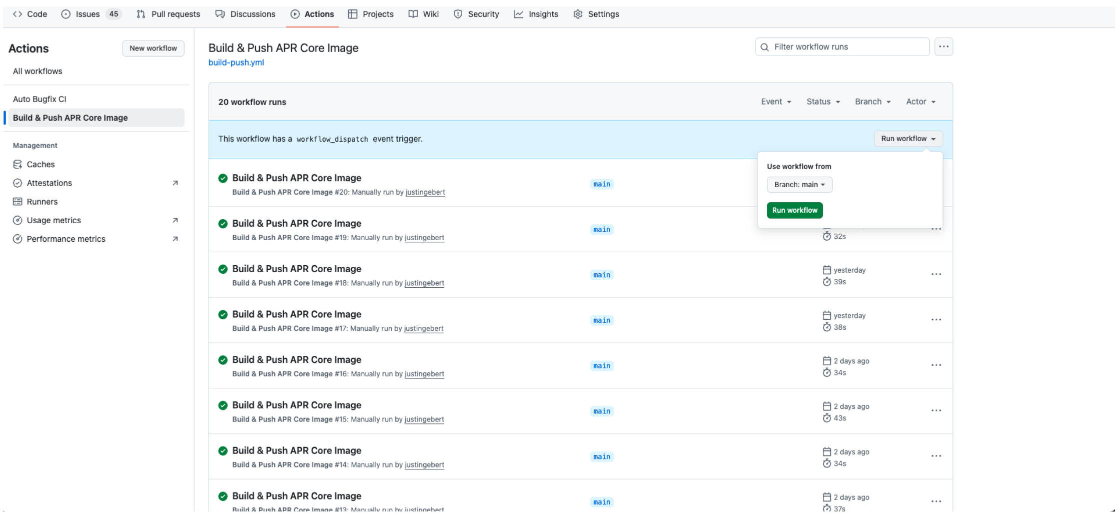

Figure 6.3.: *Manual Dispatch of APR*

When the workflow is triggered it will create a new run in the GitHub Actions tab. This execute the relevant steps previously described in 5.

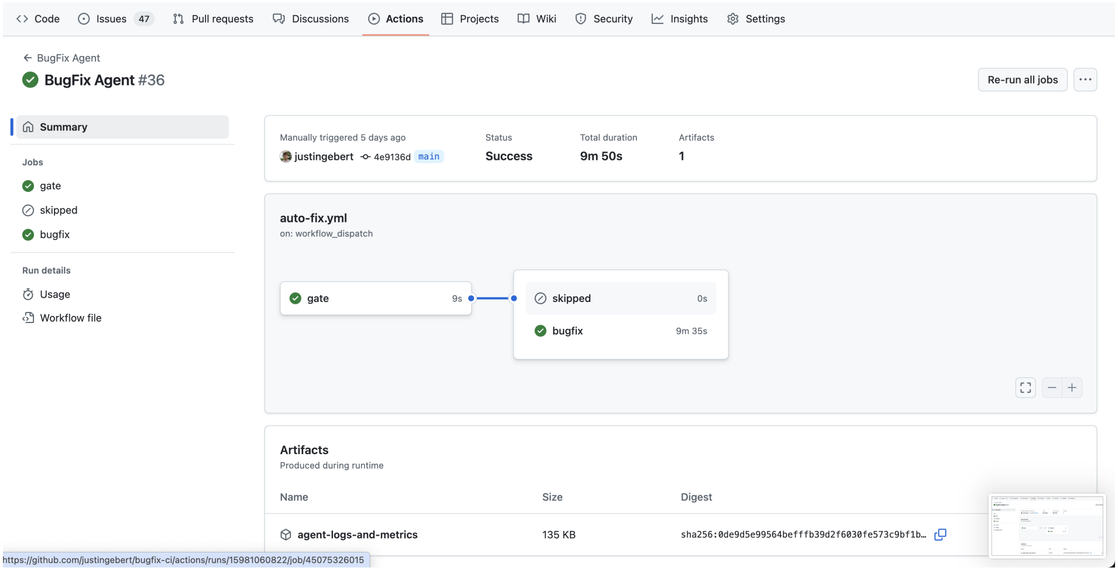

Figure 6.4.: *GitHub Action Run*

A run can produce two possible outcomes for each issue it processes. On a successful repair attempt it will create a pull request with the changes made to the codebase and the issue it is related to. This allows for easy review and merging of the changes into the main branch. The pull request will also contain information about the issue that was fixed, making it easy to track the changes made to the codebase.

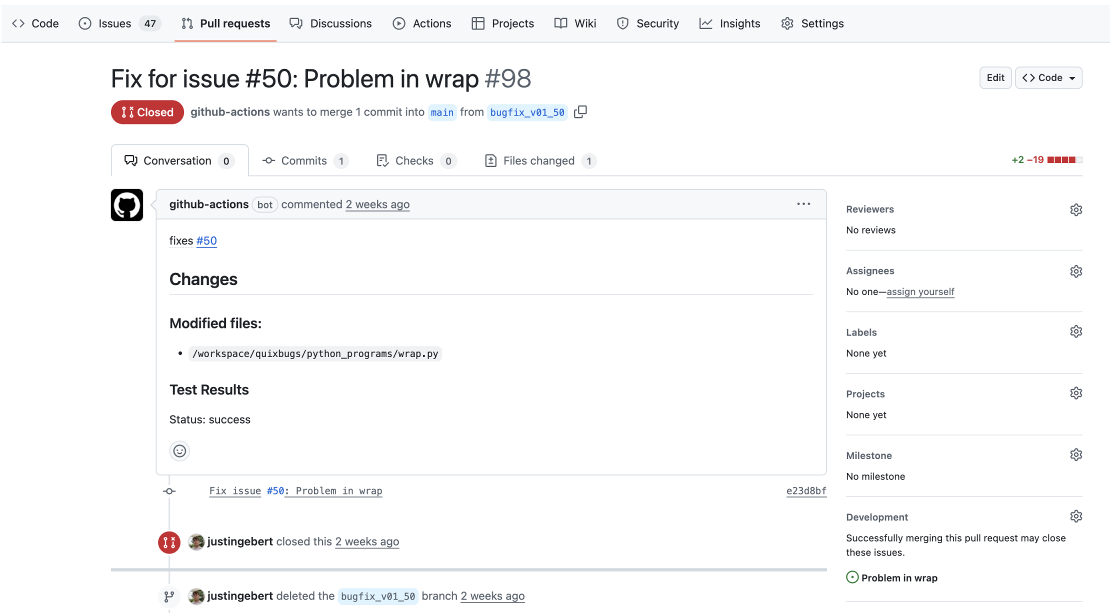

Figure 6.5.: *Resulting Pull Request*

When a repair attempt fails for an issue that is processed the failure is reported to the issue and the issues is labeled as failed so it wont be picked up again. This allows for easy tracking of issues that could not be fixed by the APR system.

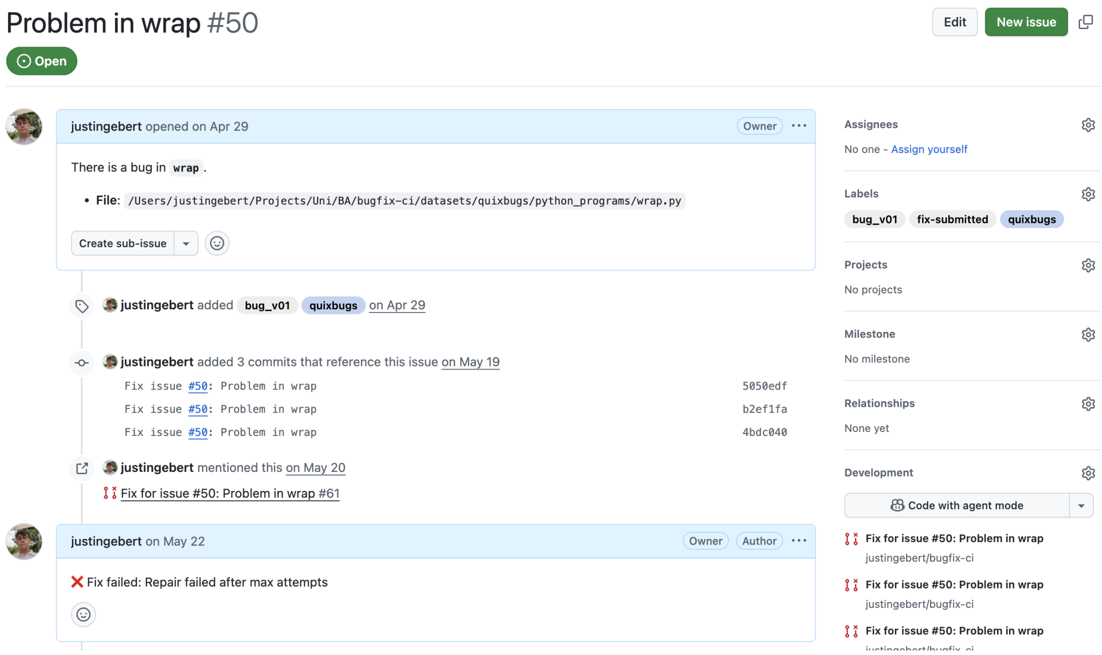

Figure 6.6.: *Failure Report*

When a new comment containing context gets added to the issue the APR system will automatically pick up the issue again. 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. 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 context are also stored as artifacts to download.

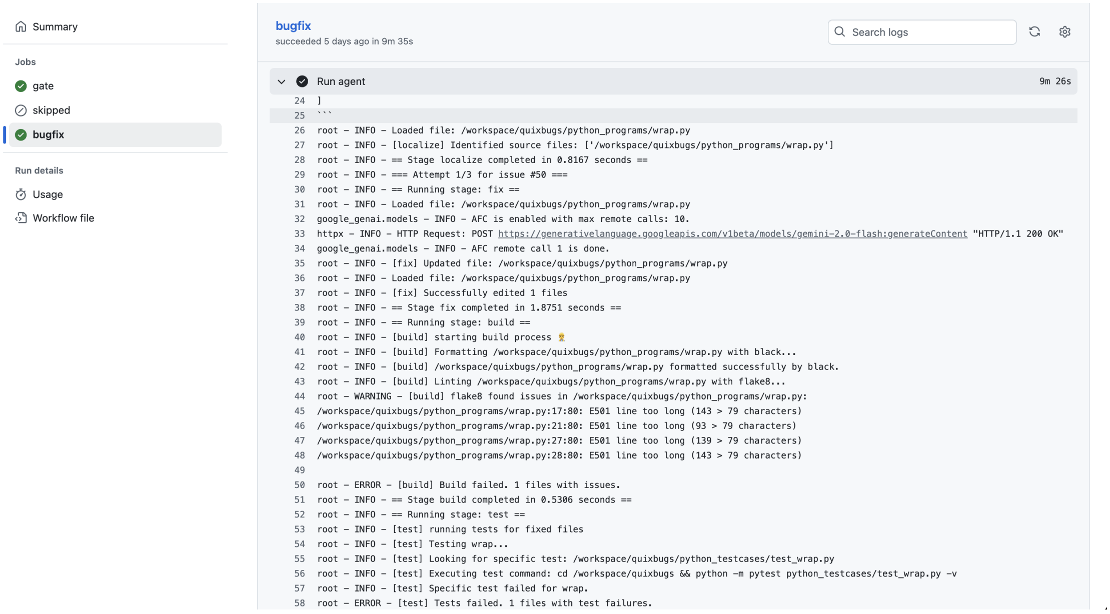

Figure 6.7.: *APR log stream*

In figure 6.8 we can see the resulting flow diagram of the APR system. The diagram shows the steps taken by the user which have an effect on the state of the system and the steps taken by the system itself.

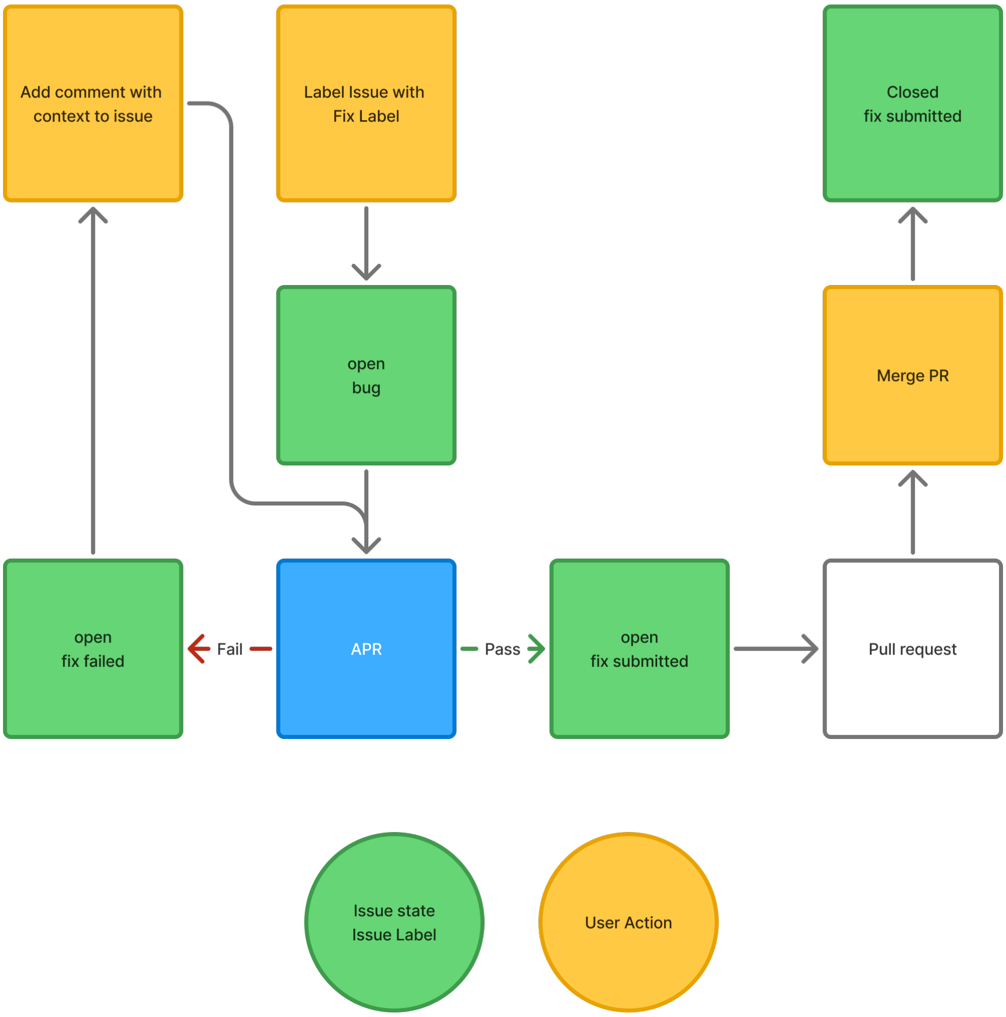

Figure 6.8.: *Resulting flow diagram*

6.2. Evaluation Results

In this section we will present the results of the quantitative evaluation of the APR prototype. This evaluation is based on the data collected and calculated for each run of the prototype 3.2.

6.2.1. Validity

- github runners have a lot of computational noise - only small set of runs.

more regarding validity of results in section 7.1.

6.2.2. Baseline of Evaluation

For the evaluation we executed the APR prototype on the repository containing 40 issues from the QuixBugs benchmark. 3.1.3. To evaluate effectiveness, performance and cost we evaluate the APR prototype using 12 different LLMs model defined in 3.1.2. This resulted in X pipelines runs each processing all 40 issues from the repository.

6.2.3. Results

The results of the evaluation are summarized in 6.1.

For testing the impact of the retry loop we collected the values with the retry loop enabled and disabled. The results are shown in 6.2.

The table shows the repair success rate, average cost and average execution time for each model used in the evaluation.

Model	**Repair Success Rate**	**Average Cost**	**Average Number of Attempts**	**Average Execution Time**
gemini-2.0-flash	92.5%	0.00	0.00	0.00s
gemini-2.5-flash-lite	95%	0.00	0.00	0.00s
gemini-2.5-flash	0.00%	0.00	0.00	0.00s
gemini-2.5-pro	100%	0.00	0.00	0.00s
gpt-4.1-nano	0.00%	0.00	0.00	0.00s
gpt-4.1-mini	0.00%	0.00	0.00	0.00s
gpt-4o	0.00%	0.00	0.00	0.00s
o4-mini	0.00%	0.00	0.00	0.00s
claude-3-haiku	0.00%	0.00	0.00	0.00s
claude-3-5-haiku	0.00%	0.00	0.00	0.00s
claude-3-7-sonnet	0.00%	0.00	0.00	0.00s
claude-sonnet-4-0	0.00%	0.00	0.00	0.00s

Table 6.1.: *Results of evaluation*

The table shows the repair success rate, average cost, average number of attempts and average execution time for each model used in the evaluation.

Complete artifacts are available in the repository in the Appendix A.

Model	Repair Success Rate	Average Cost	Average Number of Attempts	Average Execution Time
gemini-2.0-flash	92.5%	0.00	0.00	0.00s
gemini-2.5-flash-lite	95%	0.00	0.00	0.00s
gemini-2.5-flash	0.00%	0.00	0.00	0.00s
gemini-2.5-pro	100%	0.00	0.00	0.00s
gpt-4.1-nano	0.00%	0.00	0.00	0.00s
gpt-4.1-mini	0.00%	0.00	0.00	0.00s
gpt-4o	0.00%	0.00	0.00	0.00s
o4-mini	0.00%	0.00	0.00	0.00s
claude-3-haiku	0.00%	0.00	0.00	0.00s
claude-3-5-haiku	0.00%	0.00	0.00	0.00s
claude-3-7-sonnet	0.00%	0.00	0.00	0.00s
claude-sonnet-4-0	0.00%	0.00	0.00	0.00s

Table 6.2.: *Results of evaluation with retry loop enabled*

7. Discussion

In this section, we will discuss the results of the evaluation of our prototype and put it into context to answer the research questions. First evaluating the validity of our findings, the potential of our approach, its limitations, and summarize the lessons learned. Finally, we will outline a roadmap for future extensions of our work.

7.1. Validity

The results from the evaluation are very promising but still face some limitations to the validity of the results.

Because the repair process is based on LLMs, which are non-deterministic by design, executions can vary in their results. Furthermore speed and availability of the tested LLMs is highly dependant on the providers APIs which can account for varying execution times during high traffic times. In addition, the system was executed on GitHub provided GitHub Actions runners included in the GitHub free Limits. Therefore the performance metrics reflect GitHubs cloud-hosted CI environments. While allowing quick iterations and setup this limits the feasibility of absolute, execution times and costs.

The bases of evaluation is the QuixBugs benchmark which consists of 40 single line issues each in a separate file. This dataset is not fully representative of real-world software development, as it only covers a small niche from the complexity and variety of bugs that can arise in larger codebases. In addition we assume that the tests show reliable correctness of a tested program. Although QuixBugs provides extensive tests for the size of the programs, these do not guarantee full behavioral equivalence with the ground truth. Consequently, a generated patch may pass the tests while being semantically incorrect.

We partially offset these limitations by evaluating against twelve diverse LLMs that are likely to translate to larger datasets and other CI platforms. But testing on larger benchmarks like Defects4J and SWE-Bench are required before drawing conclusions about real-world effectiveness.

The results demonstrate that an LLM based automated bug fixing pipeline can be integrated into a CI workflow and achieve non-trivial repair rates at with minimal time effort and low cost. Nevertheless, the threats outlined above delimit the scope of that claim. More benchmarks and programming languages are necessary to fully validate the approach in production scale settings.

–add that knowledge cutoff is after the benchmark was publishes since the benchmark is open source it might have been in the training dataset of the models

7.2. Potentials

hand of fixing small bugs: - can take over small tasks in encapsulated environment without intervention - the workflow shows that bugs can be fixed by just adding a label to an issue

- small models can solve more problems with retrying with feedback - no description is needed to solve small issues - this concept is applicable to other python repositories - configuration makes it adjustable to different repositories and environments - similar results to other approaches -> is feasible

the results show promising reapir effectiveness of X% up the cheapest model X

- combines agentless and a bit of interactive but interactivity is limited due to timings but can resemble real world remote environment

-with small models and attempt loop makes small models pass the whole benchmark?

agent architectures produce good results epically paired with containerized environments. [2]

- the framework is model agnostic and as LLMs get better the

?? - accelerate bug fixing - lets developers focus on more complex tasks - therefor enhance software reliability and maintainability

this setup is portable, modular and extendable so it can be adaped and further tests can be made more in 7.5

7.3. Limitations

Ultimately, there are also limitations faced by the prototype and the approach in general.

As mentioned in the validity section 7.1, the system is contained to addressing small issues only. Even with small issues the timings and availability of the system is highly dependant on external dependencies like the LLM providers APIs and the provided GitHub Actions runners. Which makes this a limitation in terms of reliability and performance in a real software development cycle.

In terms of the integration with GitHub is that the system faces more limitations that come from the Github Actions environment. Runs can not be skipped and filtering logic of events with external configuration data is very limited. This results in the workflow having to execute on every issue labeling event to filter in the first job. This fills the GitHub Actions tab with runs that have success as status but do not have any bugs to fix, resulting in cluttered the run history which may lead to confusion for users.

This could be solved by migrating the APR core to a GitHub App which listens to webhook events and only triggers the workflow for relevant issues.

Additionally security and privacy concerns arise from the fact that the program repair is based on LLMs. Since issue title and description get added to the prompt that is used for repair, malicious instructions could be put into an issue. Therefore non trusted project contributors should not be able to create or edit issues. Furthermore code submitted for fixing the bug as a pull request needs to be verified and reviewed carefully before merging.

Large 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 the code and issues processed by the system, especially for private or sensitive repositories. The prototype was not tested using open source models, but is by design modular and extendable for the use of open source model. Nevertheless, copyright and licensing issues may arise when code is generated by LLMs that are based on copyrighted training data. [28, 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 LLM and the training data. - 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

- richer benchmarks - Service Accounts for better and more transparent integration - try out complex agent architectures and compare metrics and results - try out more complex bug fixing tasks - SWE bench - concurrency and parallelization of tasks - app which replies on webhook events - 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] *About Workflows*. https://docs-internal.github.com/_next/data/9uQSGns-DWbCy3Cy8blUA/en/fre pro-team%40latest/actions/concepts/workflows-and-actions/about-workflows.json?versionId=free-pro-team%40latest&productId=actions&restPage=concepts&restPage=workflows-and-actions&restPage=about-workflows. (Visited on 06/25/2025).

[23] 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).

[24] *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).

[25] 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).

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

[27] 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).

[28] 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).

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

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

[31] Abdul Sajid Mohammed et al. "AI-Driven Continuous Integration and Continuous Deployment in Software Engineering". In: *2024 2nd International Conference on Disruptive Technologies (ICDT)*. Mar. 2024, pp. 531–536. DOI: 10.1109/ICDT61202.2024.10489475. (Visited on 04/23/2025).

[32] 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).

[33] 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).

[34] 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).

[35] 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).

[36] 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).

[37] 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).

[38] 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).

[39] 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).

[40] 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).

[41] 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).

[42] 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).

[43] 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).

[44] 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).

[45] 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).

[46] 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).

[47] 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).

[48] 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).

[49] 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).

[50] 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).

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

A. Appendix

A.1. Source-Code

Github Prototype: `https://github.com/justingebert/bugfix-ci` Github Evaluation repository: `https://github.com/justingebert/quixbugs-apr`

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