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

# 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 debugging are getting 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 have direct negative impact on 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 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 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 Continous Integration (CI) pipelines. Continuous integration is the backbone of modern software development, ensuring rapid, 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 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 explains required the theoretical background and context for this thesis. First introducing fundamental concepts of software development, the software development lifecycle (SDLC), Continuous Integration (CI), and software project hosting platforms. The second part explores GenAI/LLMs and their rising role in software development practices. The third part showcases the evolution and state of Automated Program Repair (APR) and explores existing approaches.

## 2.1. Software Development

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 software project 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].

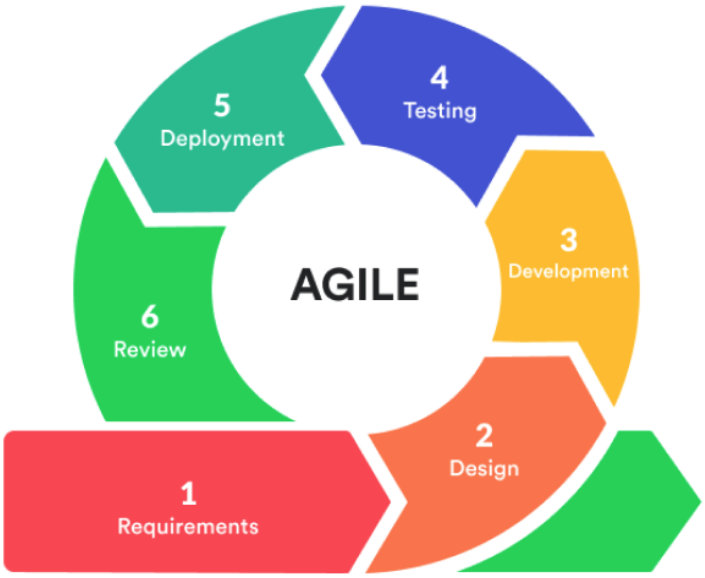


**Figure 2.1.:** *Agile software development lifecycle*

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].

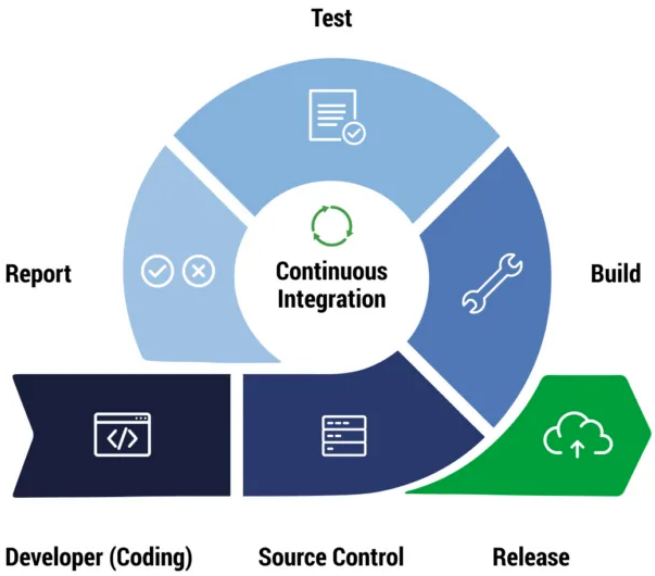


**Figure 2.2.:** *Continuous Integration cycle*

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.

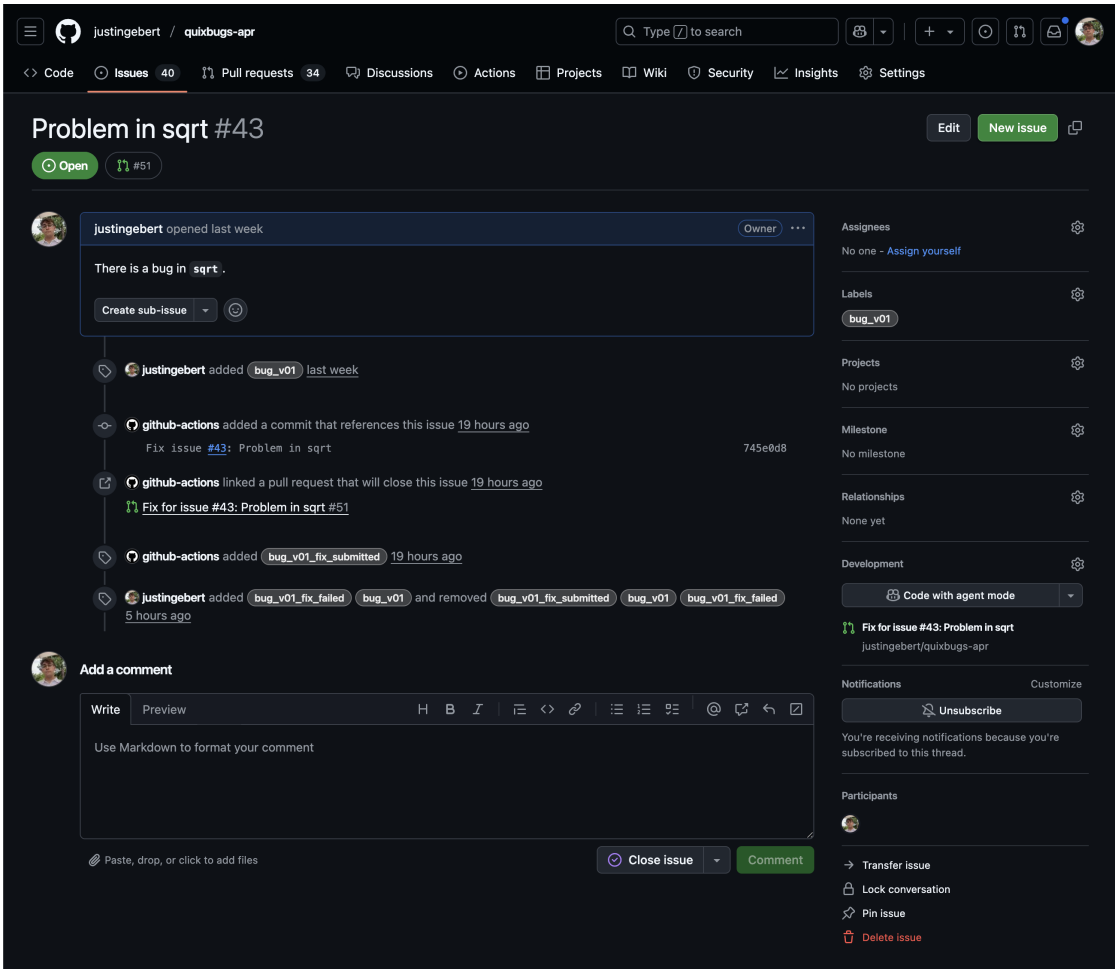


**Figure 2.3.:** *Example of a GitHub Issue*

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

[1] TODO explain format

and manage their CI processes directly within their repositories.

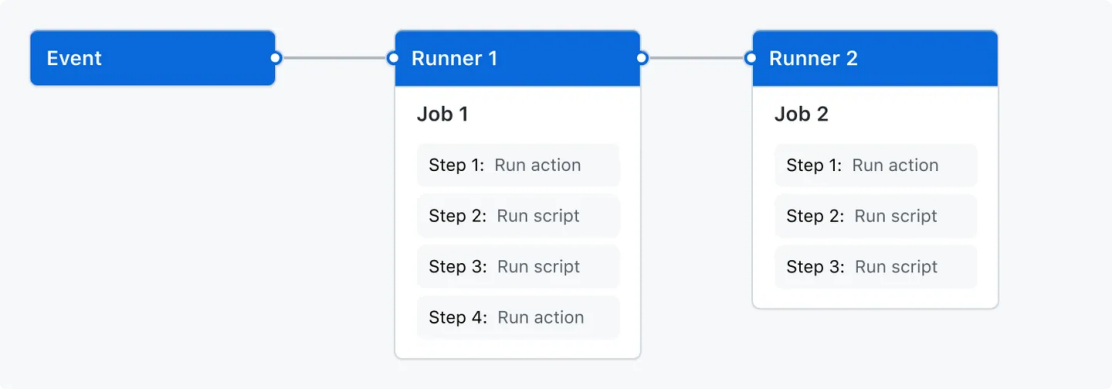


**Figure 2.4.:** *Building blocks of a Github Action*

## 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 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?

For answering these questions we streamlined this process into three phases preparation, implementation/usage and evaluation. The process is visualized below in Figure 3.1.

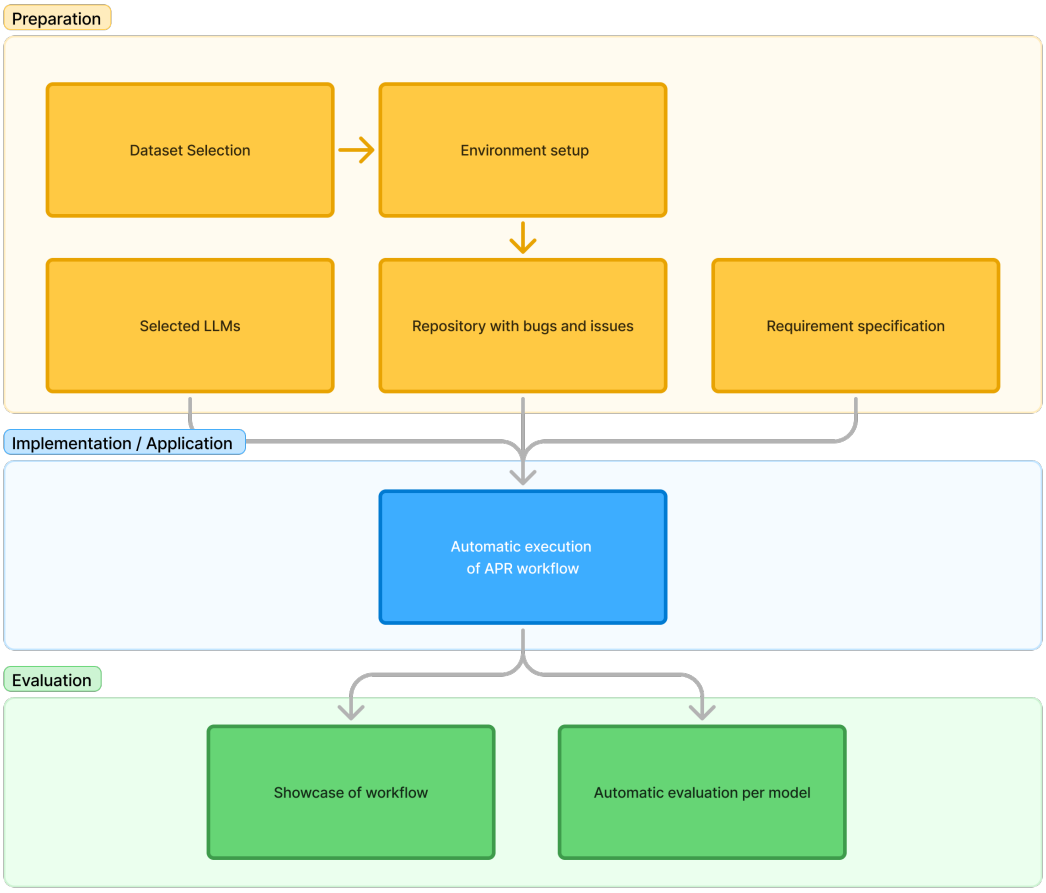


**Figure 3.1.:** *Thesis methodology approach*

In the preparation 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 evaluated the self developed prototype by using the defined evaluation metrics 3.3 collected during and after the execution of the APR system. Furthermore we will showcase the resulting workflow of using the system in the prepared 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, 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 the evaluating the effectiveness 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 bugs in Python[1]. It consists of 40 individual files, each containing an algorithmic bug. Each bug is caused by 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 challenging problems for developers [46], this means it enables us to evaluate if our system can take over the cognitive demanding task of fixing small bugs without developer intervention.

Additionally 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 a selected pool of LLM models. Focusing on smaller models for this benchmark. Small models have fast response times and produce little costs. We will compare a selection of recent (point 11.07.2025) models from well known providers (Google, OpenAI, Anthropic). The models are selected to cover a range of capabilities and costs with focus on lower tier models, allowing us to evaluate the performance and cost-effectiveness of the APR system. The following were models selected and will be used for the evaluation:

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

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

| Model Name | Publisher | Context Window Size in Tokens | Cost per 1M Tokens | Provider Description |
|---|---|---|---|---|
| **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-3-5-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 that captures only the title of the Problem with a minimal description. The generated issues serve as the entry point and communication medium to the APR system. Figure 2.3 shows an example of a generated GitHub issue.

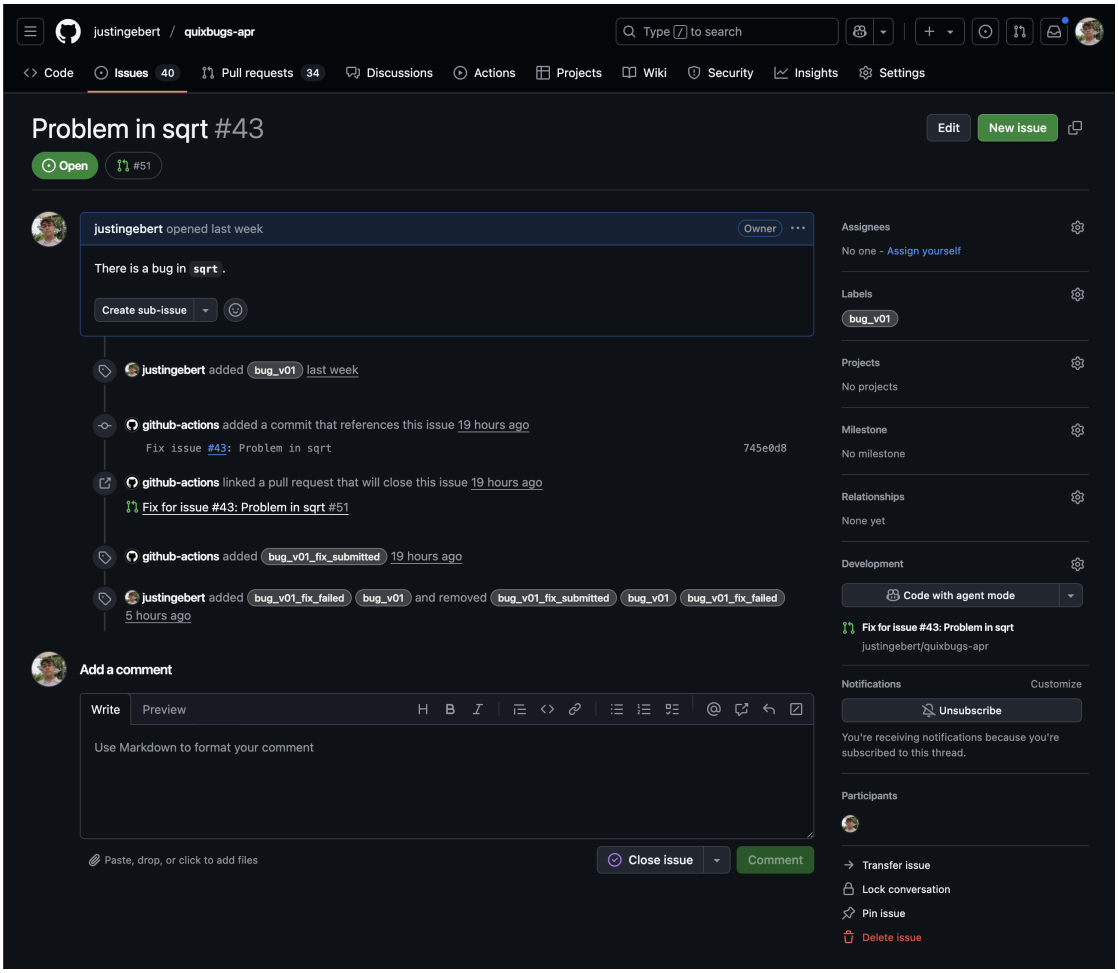


**Figure 3.2.:** *Example of a generated GitHub Issue*

### 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 [51]. According to the INVEST principles, each requirement was formulated to be independent, negotiable, valuable, estimable, small, and testable. With this framework we constructed 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 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:

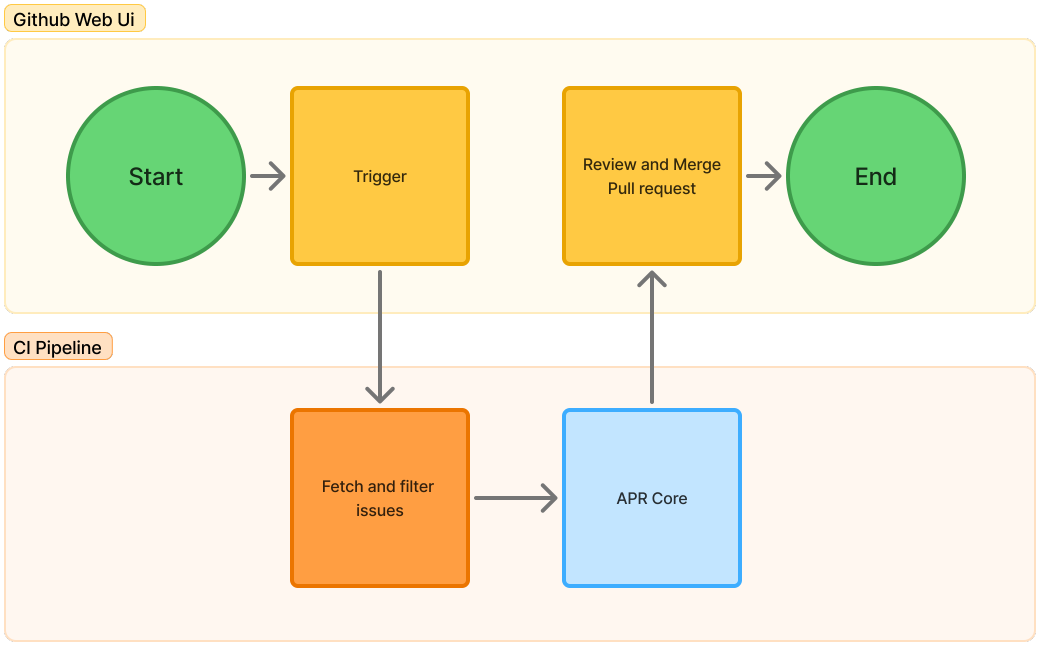


**Figure 3.3.:** *Overview of the APR Pipeline*

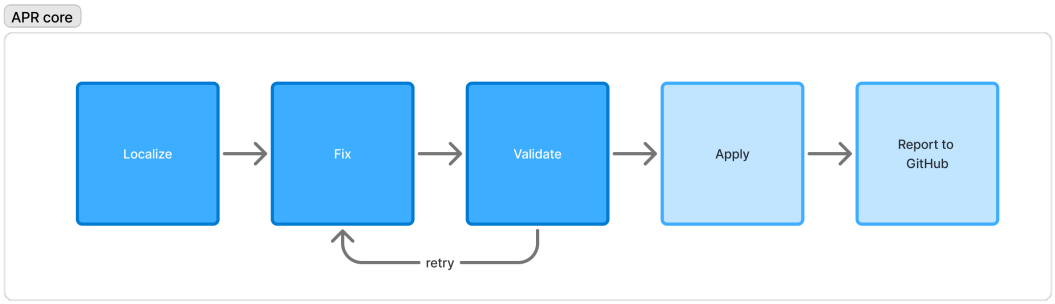


**Figure 3.4.:** *Overview of the APR Core*

When the system is in place in a repository the pipeline can automatically be triggered by the configured triggers. This will execute the pipeline on a GitHub Action runner. In the first pipeline job relevant issues are fetched and filtered for further processing. Filtered issues are then passed to the Dockerized APR Core which contains the main bug fixing logic. The APR core talks to 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 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 maximum attempts a failure is reported to the issue.

## 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. Using Github Actions Continuous Integration abilities, with the prepared QuixBugs repository as a base. We evaluate all the selected LLM models 3.1.2 in the context of the APR system. The evaluation is based on the 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 abilities in repairing software bugs. These metrics will provide insights into the system's effectiveness, performance, reliability, cost and overall impact on the software development lifecycle.

For evaluating the effectiveness of the LLM models used and 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 issues is considered successfully repaired when the validation passes with correct syntax and the related test suite passes all tests.

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

For evaluating performance, we will analyze collected and fine grained timings of an issue repair attempt. 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, since 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 and issue repair process 3.4. Using this data we calculate per model run results listed in table 3.5 which will help in answering the RQ2.

| **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*

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 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 | $\frac{Number of Successful Repairs}{Total Issues Processed}$ |
| Average Number of Attempts | $\frac{Total Attempts}{Total Issues Processed}$ |
| Average Execution Time | $\frac{Total Execution Time}{Total Issues Processed}$ |
| Average Tokens Used | $\frac{Total Tokens Used}{Total Issues Processed}$ |
| Average Cost | $\frac{Total Cost}{Total Issues Processed}$ |
| Average Stage Execution Time | $\frac{Total Stage Execution Time}{Total Stages Processed}$ |
| Average Stage Outcome Success Rate | $\frac{Number of Successful Stages}{Total Stages Processed}$ |

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

# 4. Requirements

To guide the implementation of the prototype we developed the following requirements to help design the prototype. These requirements allow for better planning and prioritization during implementation and tracked progress. Following the INVEST model [51] we constructed functional 4.1 and nonfunctional 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 A appendix.

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. The complete code implementation can be on GitHub listed in Appendix A

**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 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 |

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

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 logic is a predefined flow which is makes use of multiple stages and tools.

At 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 used by a stage to perform a specific task in the bug fixing process and gets returns with the added context. The cores' stages are Localize, Fix, Build and Test.

The repair process starts with the localization stage. This stage tries to find the files needed from the codebase to fix the bug, using the configured LLM Model via the providers SDK[3]. The localization 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. 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

```
5
6       Issue #{issue['number']}: {issue['title']}
7       Description: {issue.get('body', 'No description provided')}
8
9       Repository files:
10      {json.dumps(repo_files, indent=2)}
11
12      Return a JSON array containing ONLY the paths of files that need to be
        modified to fix this issue.
13      Example: ["path/to/file1.py", "path/to/file2.py"]
14      """
```

**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 Model API to generate a fix for the issue in the localized files. The prompt for requesting the fix contains the issue details and file names with file content. The response is expected to contain a list of edits for each file. The LLM can also specify that no changes need to be made in a file. 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.

```
1  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."
2
3  base_prompt = f"""
4      The following Python code files have a bug. Please fix the bug across all
        files as needed.
5
6      {files_text}
7
8      Please provide the complete, corrected source files. If a file doesn't
       need changes, you can indicate that.
9      For each file that needs changes, provide the complete corrected file
       content.
10     Format your response as:
11
12     === File: [filepath] ===
13     [complete file content or "NO CHANGES NEEDED"]
14
15     === File: [filepath] ===
16     [complete file content or "NO CHANGES NEEDED"]
17     """
```

**Listing 5.3:** *Repair Prompt*

For validating the generated edits 2 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/maintenance rules [4]. To archive this the code is first formatted using the Python formatter Black[5] and secondly 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 will jump back to the fix stage with a new attempt.

For a new attempt additional feedback is generated using the previous code and stage results with and attached to the prompt. When the maximum number of attempts is reached and the code does not pass testing an unsuccessful repair is reported to the issue by creating a comment using the Github API.

With successful validation and tests the issue is marked as a successfully repaired. The file changes are committed and pushed to the remote repository. A Pull Request is created to merge the issue branch in to the main branch. This Pull Request holds detailed file diffs and links the issue.

During execution the APR Core logs every action, which can be used for debugging 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 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 illustrates the functionalities and tools used in the APR Core.

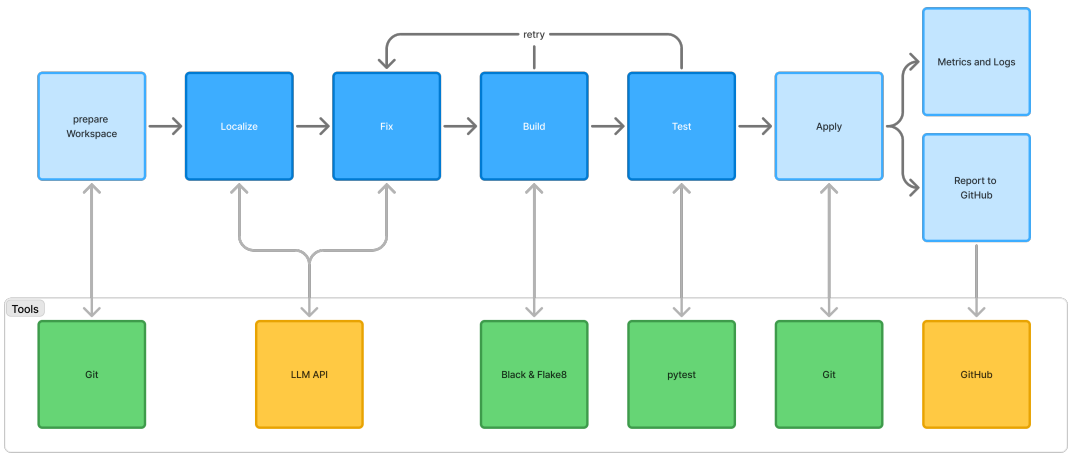


**Figure 5.1.:** *APR Core Logic*

[4] todo
[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 takes away 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 is executed two different ways:

- Batch processing by fetching all issues with state for repair. This can be triggered by a manual dispatch ("workflow_dispatch") or scheduled execution ("cron").
- Process a single issue from the event. The issue is passed from the ("issue_labeld") event when labeled with the configured labels or from ("issue_comment") when extra information is added issue in form of a comment.

The trigger event information gets passed as environment variables to the first job named "gate". This uses the data to determine if the issue should be processed or skipped using a python script (filter_issues.py). This script needs be placed accessable for the workflow file. It checks labels of the issue and resolves its state to determine if the issue is relevant for the APR process. If no issues pass "gate" the job "skipped" is executed, which simply logs that no issues were found and exits the workflow run. In case issues pass the "gate" the "bugfix" job is started. This job is responsible for executing the APR Core logic. It provides necessary prerequisites (see 5.1) to start a container using the latest APR Core Docker Image which performs the 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 issues on GitHub.

As the last step of "bugfix" is uploading the logs and metric files of the APR core as artifacts, which makes 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

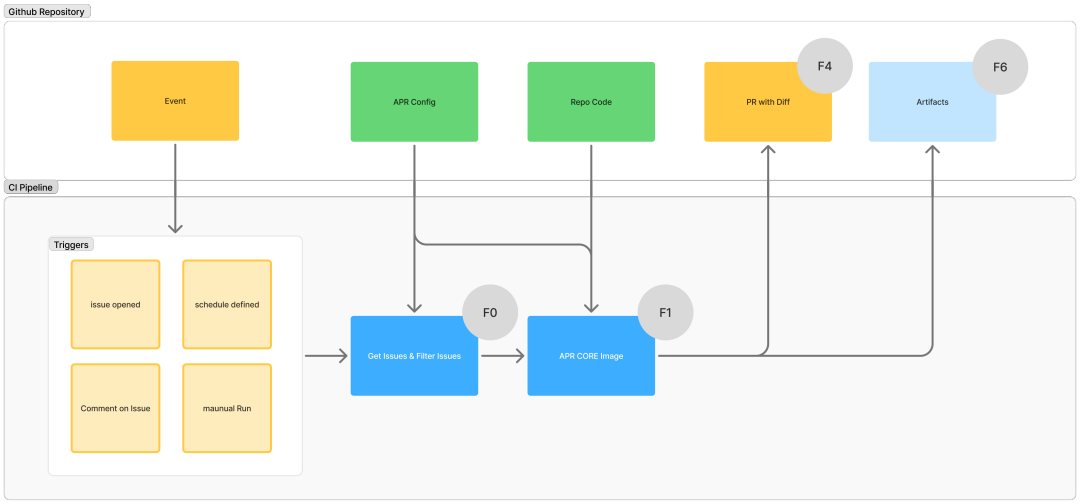


**Figure 5.2.:** *APR Core Logic*

For using this integration in a repository, the workflow file needs to be placed in the '.github/workflows' directory of the repository along with the 'filter_issues.py' in '.github/scripts'. With this in place an "LLM_API_KEY" needs to 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 needs to be granted permissions to create pull requests and write 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

For making the system easily adjustable the APR Core and the CI Pipeline can be configured using a YAML configuration file. The configuration is optional when no configuration is in place the system will use default configuration. A custom configuration must be named "bugfix.yml" and must be placed at the root of the repository. This allows for easy customization of the system without changing the code itself. The configuration read by system components during execution. Table 5.2 lists the available configuration fields with a description.

**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. |

## 5.3. Requirement Validation

The following sections outline how each requirement was satisfied.

# 6. Results

In the following section we will present our results our implementation and evaluation. We implemented a working prototype for accessing how to integrating 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 bases.

## 6.1. Showcase of workflow

Setting up the APR system in a repository is archived by adding 2 files to the ".github" directory of the repository. As seen in **??** the required files are the ".github/workflows/auto-fix.yml" and ".github/scripts/filter_issues.py". With this the system is ready to operate. Additionally a optional configuration file can be added to the root, this ".bugfix.yml" file can be used to overwrite the LLM model used, max attempts and naming conventions of labels and branches. Furthermore a "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 automated fixed in two different ways.

Processing a single issue right away when the issue is created and labeled with the "bug_v01" shown in Figure 6.1. This allows for fast feedback and quick bug fixing at issue creation and triage [1].

[1] explain triage

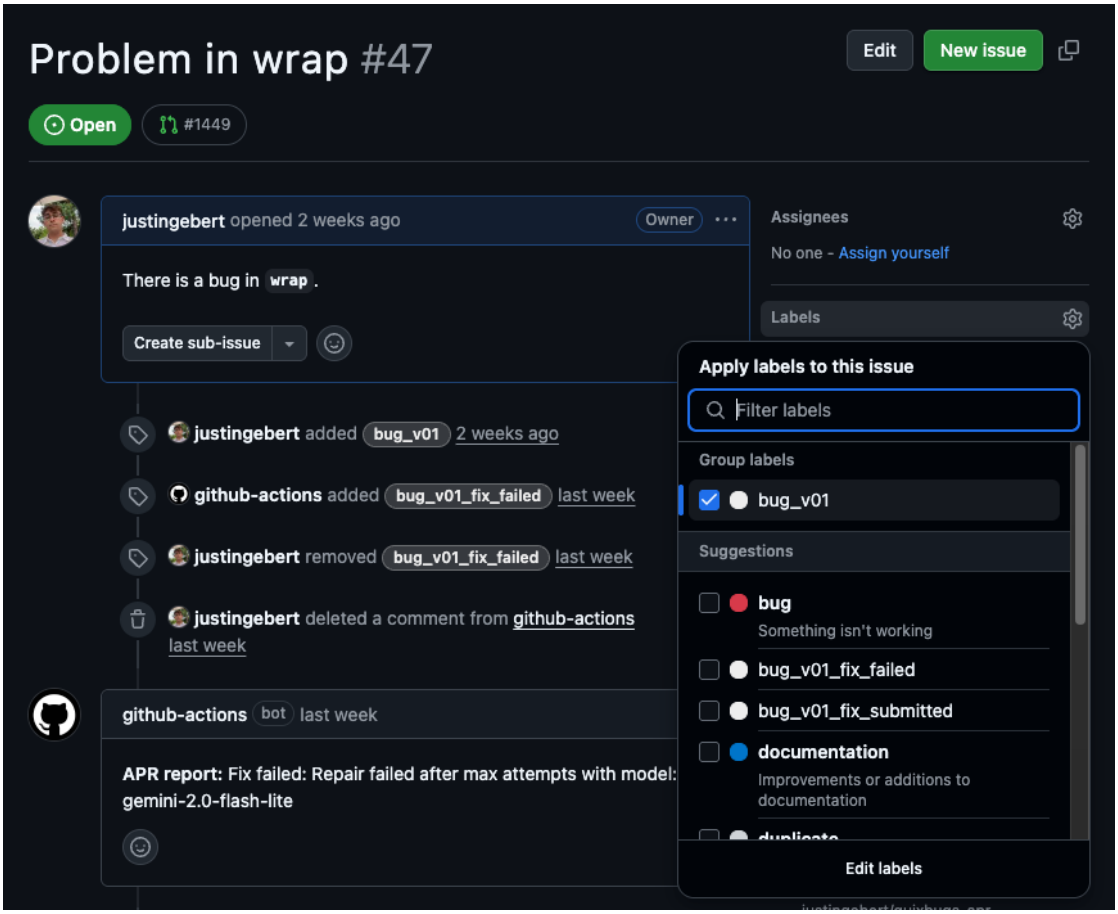


**Figure 6.1.:** *Trigger automatic fixing for single issue*

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 for bug fixing, where issues are processed in batches.

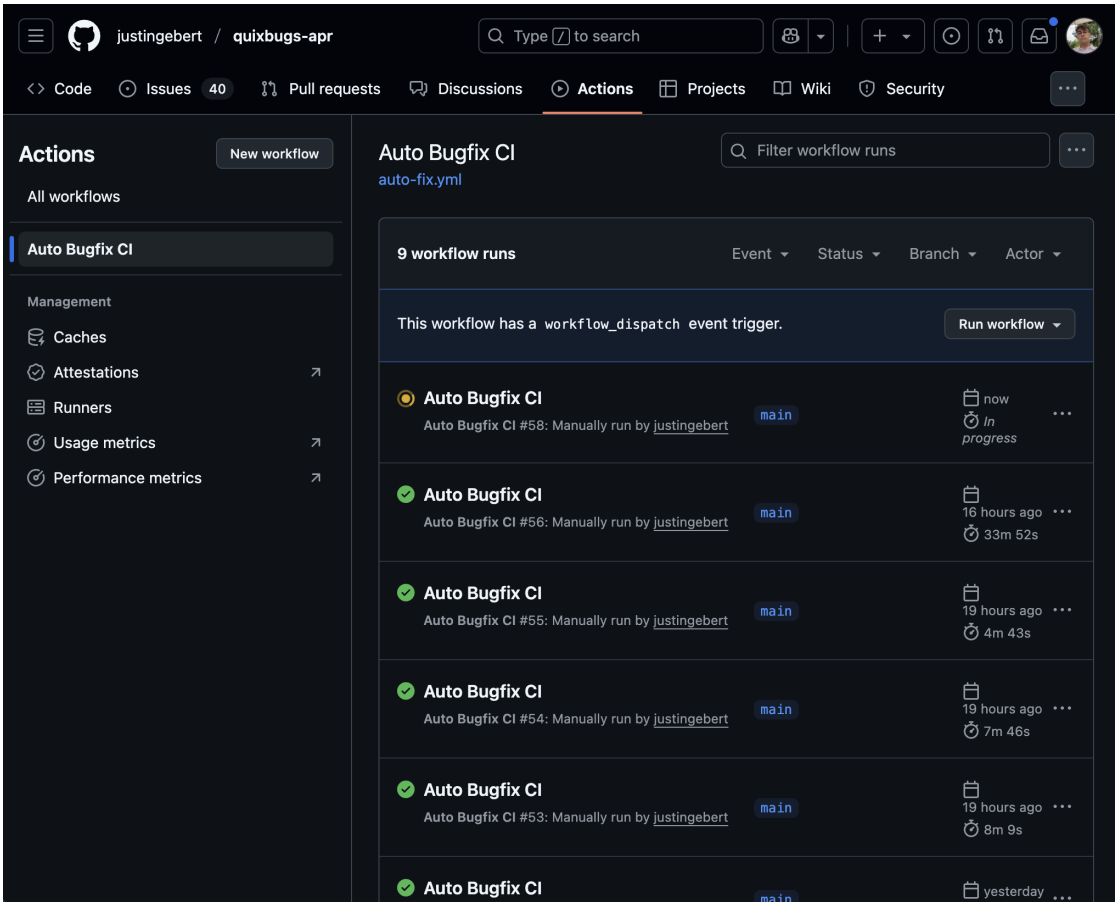


**Figure 6.2.:** *Manual Dispatch of APR*

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.

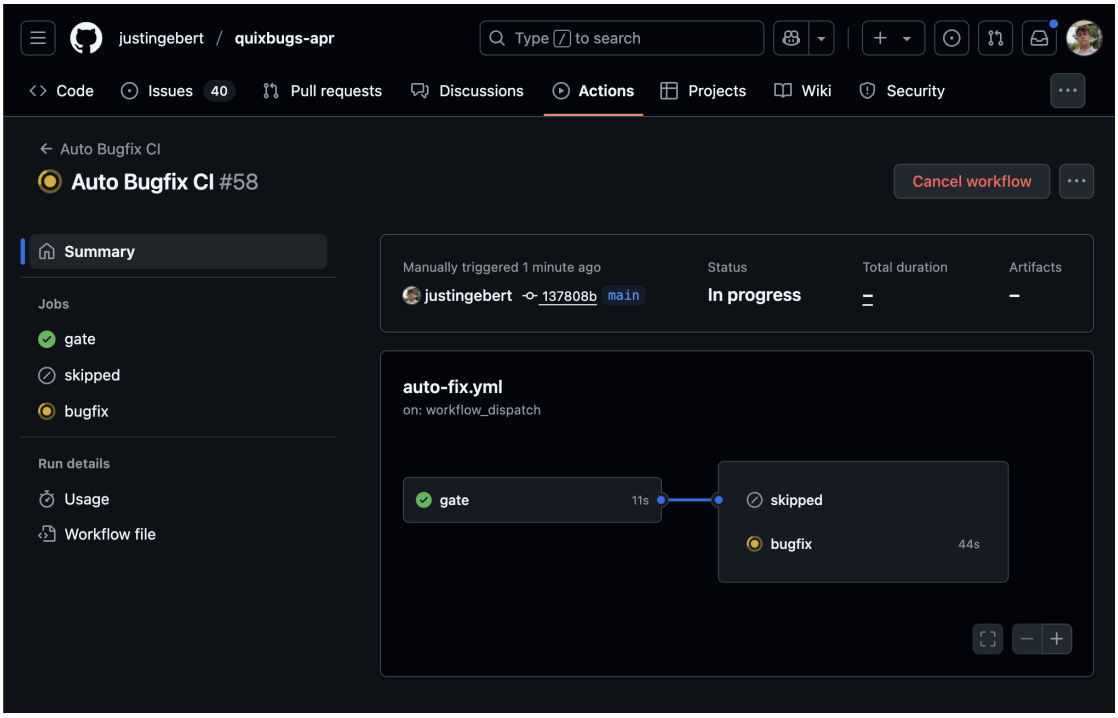


**Figure 6.3.:** *GitHub Action Run*

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 the issue linked to it. The created Pull Requests allow code review and merging of fixes changes into the main branch. When merged the issue will automatically closed. Figure 6.4 shows an example of a resulting Pull Request created by the APR system.

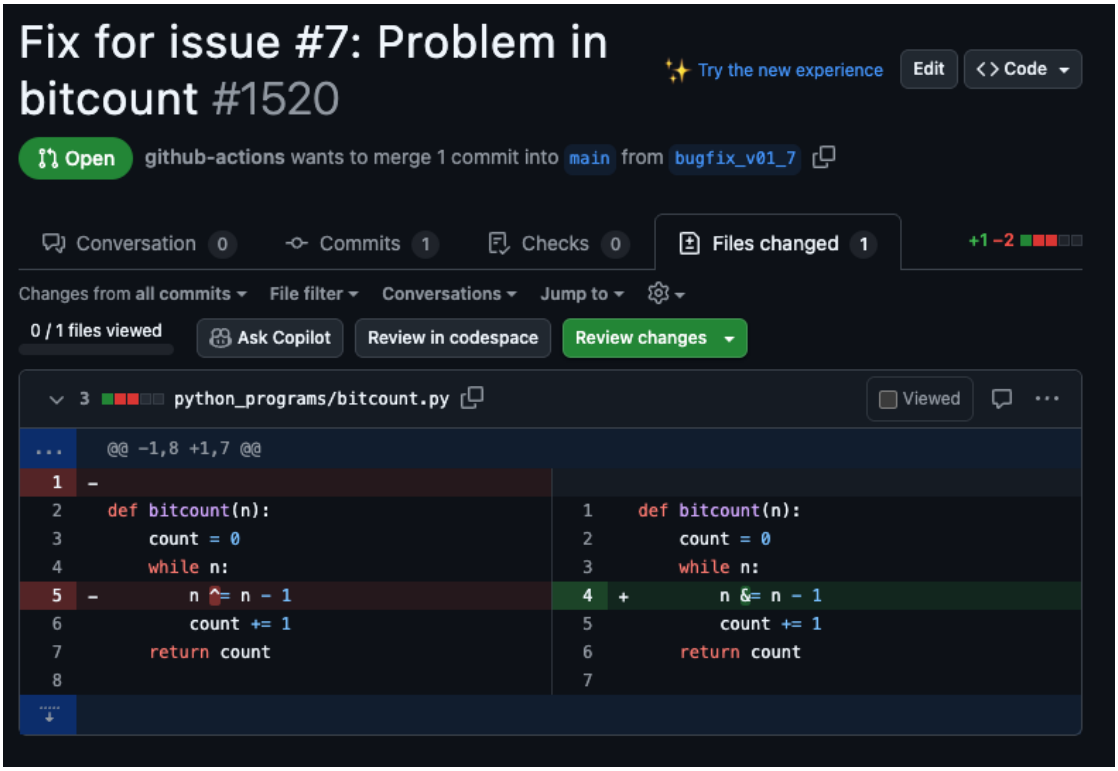


**Figure 6.4.:** *Resulting Pull Request*

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

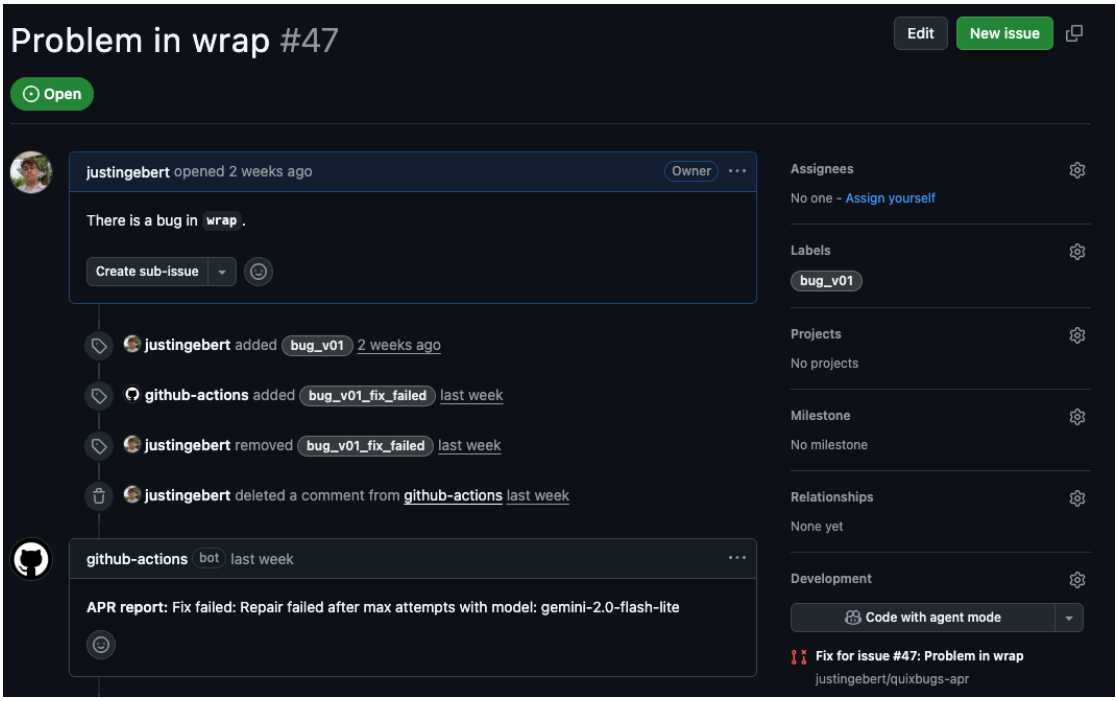


**Figure 6.5.:** *Failure Report*

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.

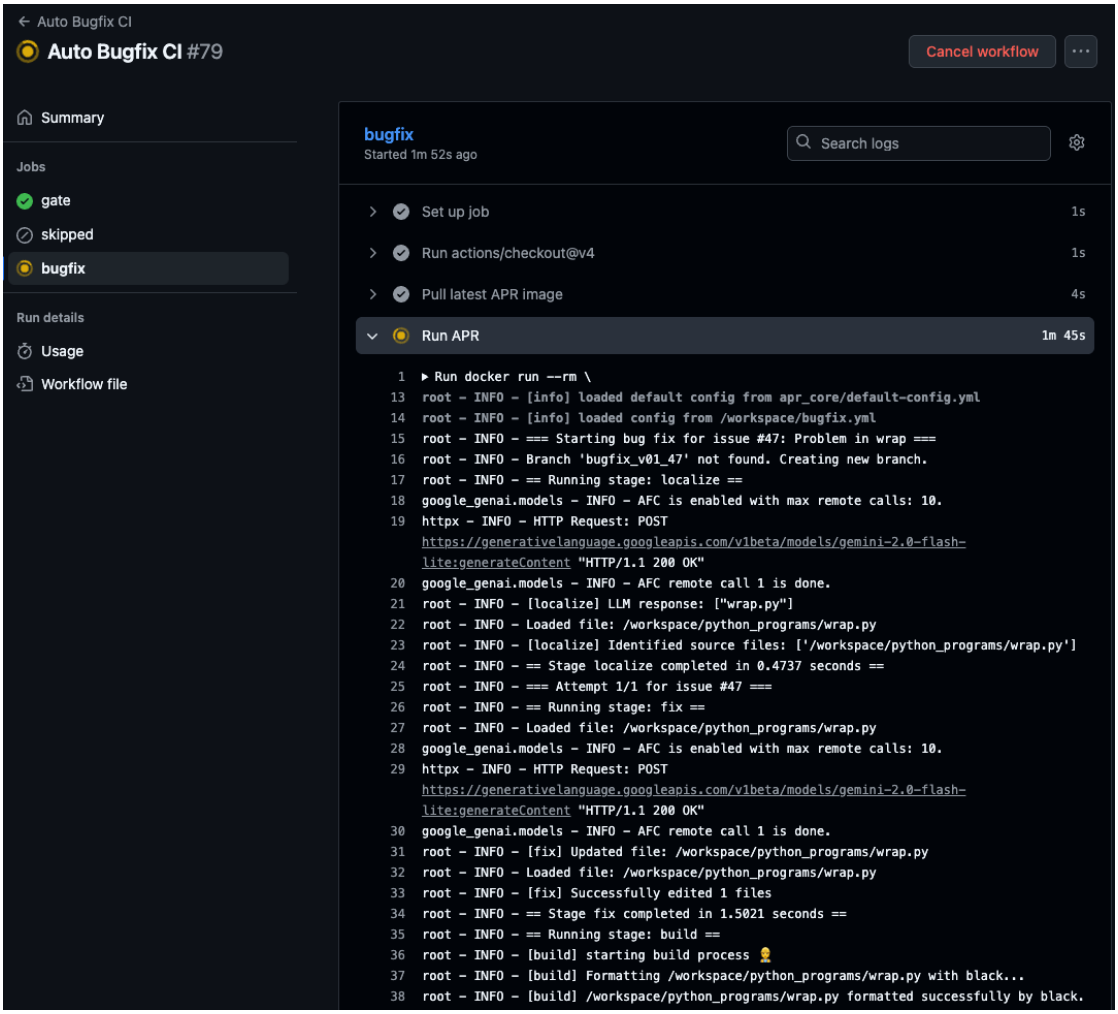


**Figure 6.6.:** *APR log stream*

Figure 6.7 visualized the resulting flow of the APR system. The diagram shows the relation between user actions (COLOR), repair success (COLOR) and the steps taken by the system.

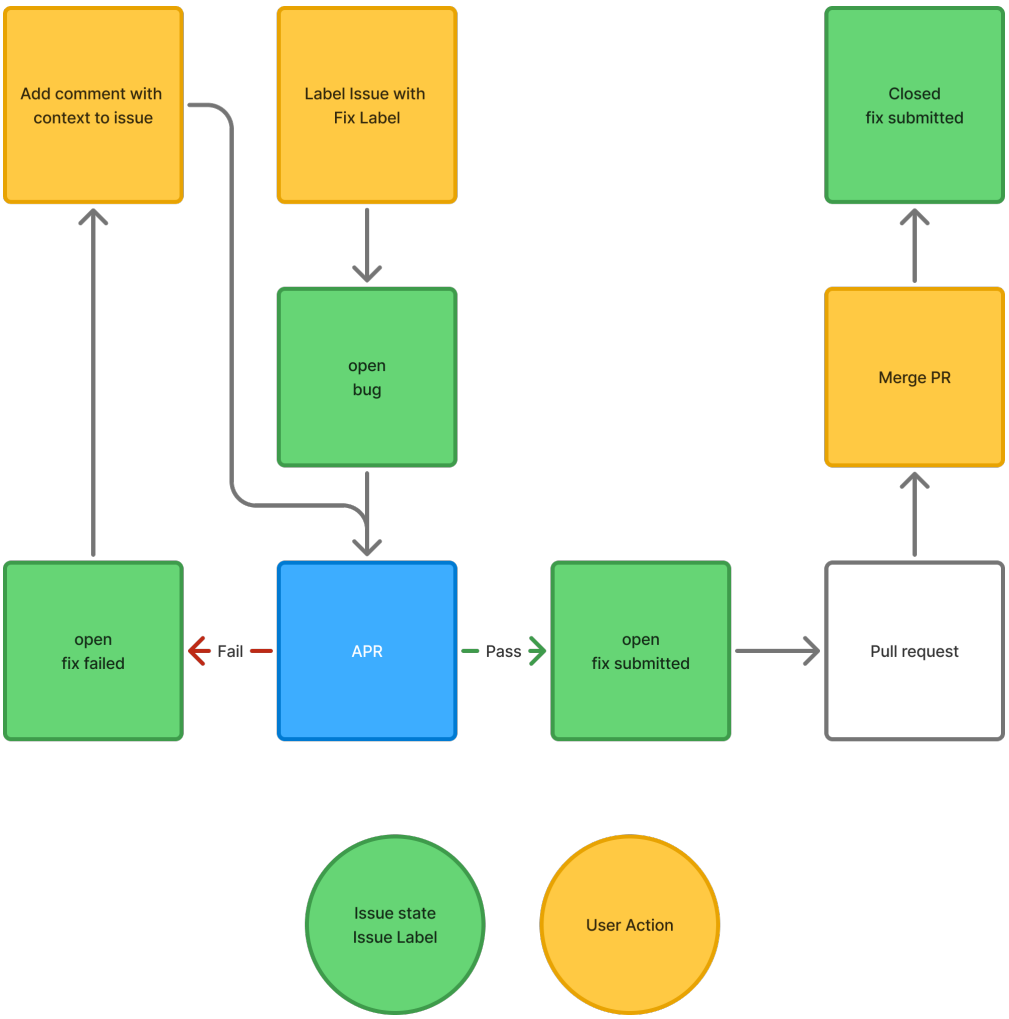


**Figure 6.7.:** *Resulting flow diagram*

## 6.2. Evaluation Results

In this section we present the results of the quantitative evaluation of the implemented APR prototype. This evaluation is done using the collected and calculated data for each run of the prototype. How this data was collected and calculated is described in 3.2.

### 6.2.1. Validity

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

We asses more threats to validity in section 7.1.

### 6.2.2. 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 LLMs models defined in 3.1.2. All models where tested with one attempt per issue to evaluate zero shot performance and with a retry loop enabled to compare a few shot performance.

### 6.2.3. Results

The following tables tables 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 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 | 11.33 |
| **gemini-2.0-flash** | 72.5% | $0.0002 | 10.81s |
| **gemini-2.5-flash-lite** | 87.5% | $0.0002 | 6.36s |
| **gemini-2.5-flash** | 90.0% | $0.009 | 50.16s |
| **gemini-2.5-pro** | 95.0% | $0.0713 | 70.09s |
| **gpt-4.1-nano** | 62.5% | $0.0002 | 14.88s |
| **gpt-4.1-mini** | 92.5% | $0.0006 | 18.75s |
| **gpt-4.1** | 92.5% | $0.0033 | 12.93s |
| **o4-mini** | 87.5% | $0.0064 | 57.05s |
| **claude-3-5-haiku** | 2.5% | $0.0024 | 23.1s |
| **claude-3-7-sonnet** | 87.5% | $0.0069 | 21.78s |
| **claude-sonnet-4-0** | 92.5% | $0.0102 | 28.93s |

**Table 6.1.:** *Zero shot evaluation results*

| Model | Repair Success Rate | Average Cost | Average Execution Time |
|---|---|---|---|
| **gemini-2.0-flash-lite** | 82.5% | $0.0002 | 19.97s |
| **gemini-2.0-flash** | 82.5% | $0.0004 | 15.62s |
| **gemini-2.5-flash-lite** | 87.5% | $0.0003 | 10.80s |
| **gemini-2.5-flash** | 92.5% | $0.01362 | 71.37s |
| **gemini-2.5-pro** | 100% | 0.00 | 0.00s |
| **gpt-4.1-nano** | 70.0% | $0.0003 | 34.85s |
| **gpt-4.1-mini** | 97.5% | $0.0043 | 24.42s |
| **gpt-4.1** | 97.5% | 0.0041 | 14.76s |
| **o4-mini** | 100% | 0.0075 | 60.98s |
| **claude-3-5-haiku** | 15.0% | $0.0075 | 65.18s |
| **claude-3-7-sonnet** | 95.0% | $0.0087 | 24.39s |
| **claude-sonnet-4-0** | 92.5% | $0.011 | 30.13s |

**Table 6.2.:** *Few shot evaluation results*

The full set of resulting data can be found the in “apr_evaluation” directiory 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 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. Since the pipeline way only run once per model these results are not fully representative of the models performance. Furthermore speed and availability of the tested LLMs is highly dependant on the providers APIs which can account for varying execution times or even failures during high traffic times. Costs are calculated based on token usage reported by the providers API responses, unfortunately this is not accurate as the providers are intransparent with actual token counts [52]. This means that the reported costs and execution times are not fully accurate across providers and should be interpreted with caution. 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 evaluation is based on 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 area of the complexity and variety of bugs that can arise in larger codebases. Moreover 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 fix 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 threats outlined above delimit the claim for problems outside the scope. Further

testing on other benchmarks and programming languages are necessary to fully validate the approach in production scale settings. Nevertheless, 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.

–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

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 the agile lifecycle by automating the design, development and testing stages in the lifecycle for bugs, 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 6.2 show that the prototype can achieve a repair success rate of up to 100% on the QuixBugs benchmark with the best performing models (X, Y, Z). 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 100% on the QuixBugs benchmark, while smaller models like gemini-2.5-flash-lite and gpt-4.1-mini achieve a repair success rate of 95% and 90% respectively. This shows that with zero shot smaller models can achieve good results, but larger models are more effective in fixing bugs.

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

[1] zero shot prompting

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 showing similar repair success rates on the QuixBugs benchmark while providing an integrated approach with transparent cost and performance insights. – add which ones, cano1killbugs, ...

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 approach taken in general.

As mentioned in section 7.1, the system is constrained 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 LLMs. - The apis of llm provders can be unreliable espically 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

- cost, time and repair success could be optimized - test different prompts - 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 - dig further into

collected data, see where it went wrong in the patches or localization - small model with fallback to larger model - more complex prompts and prompt engineering - token usage can be optimized - further analyze the collected data, we collected a lot of data which can be used for more insights

# 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/free-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.

[52] 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

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