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 ...

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 [**houLargeLanguageModels2024**, **puvvadiCodingAgentsComprehensive2025**].

Despite these advances, bug fixing remains a challenging and resource intensive task, often negatively perceived by developers [**winterHowDevelopersReally2023**]. It can cause frequent interruptions and context switching, resulting in reduced developer productivity [**vasilescuSkyNotLimit2016**]. Software bugs have direct impact on software quality by causing crashes, vulnerabilities or even data loss [**tihanyiNewEraSoftware2024**]. 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 [**CostPoorSoftware**].

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 [**houLargeLanguageModels2024**] which typically involves localization, repair, and validation [**zhangEmpiricalStudyFactors2012**, **leeUnifiedDebuggingApproach2024**, **xiaAgentlessDemystifyingLLMbased2024**, **zhangPATCHEmpowe**
wangEmpiricalResearchUtilizing2025]. 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 [**xiaAgentlessDemystifyingLLMbased2024**, **liuMarsCodeAgentAInati**
yangSWEagentAgentComputerInterfaces2024, **sobaniaAnalysisAutomaticBug2023**, **xiaAutomatedProg**
huCanGPTO1Kill2024].

However, existing APR approaches are often complex and require significant computational resources [**rondonEvaluatingAgentbasedProgram2025**], making them less suitable for budget-constrained environments or individual developers. Additionally, the lack of integration with existing software development lifecycles and workflows limits their practical applicability in real-world development environments [**chenUnveilingPitfallsUnderstanding2025**, **liuMarsCodeAgentAInative2024**].

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. [**ugwuezeContinuousIntegrationDeployment2024**] 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 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?

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 [**rupareliaSoftwareDevelopmentLifecycle2010**].

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[**rupareliaSoftwareDevelopmentLifecycle2010**].

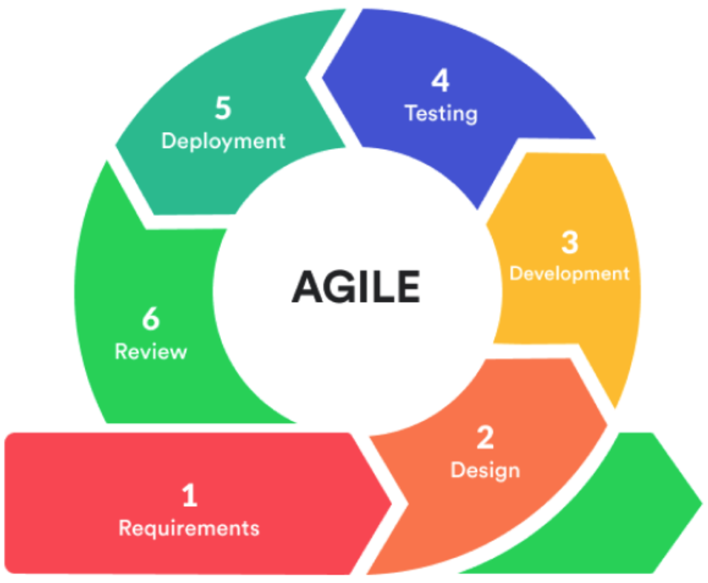


Figure 2.1.: *Agile Software Development Lifecycle*

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 [**ruparelia SoftwareDevelopmentLifecycle2010**].

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 [**ugwuezeContinuousIntegrationDeployment2024**]. 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 [**ugwuezeContinuousIntegrationDeployment2024**].

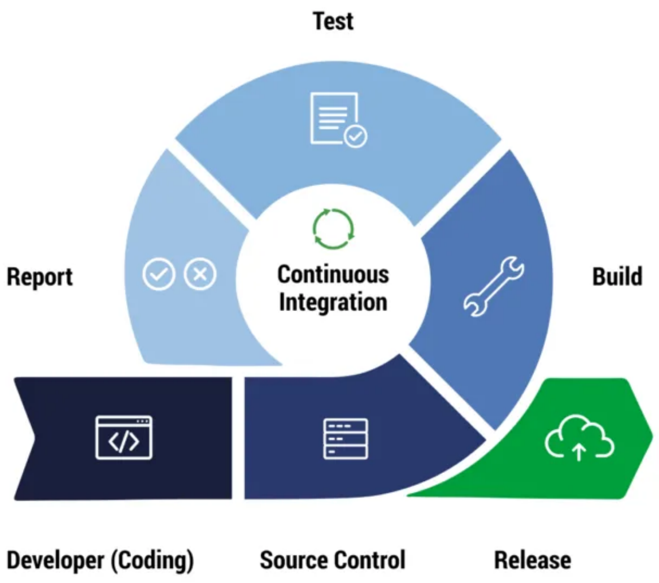


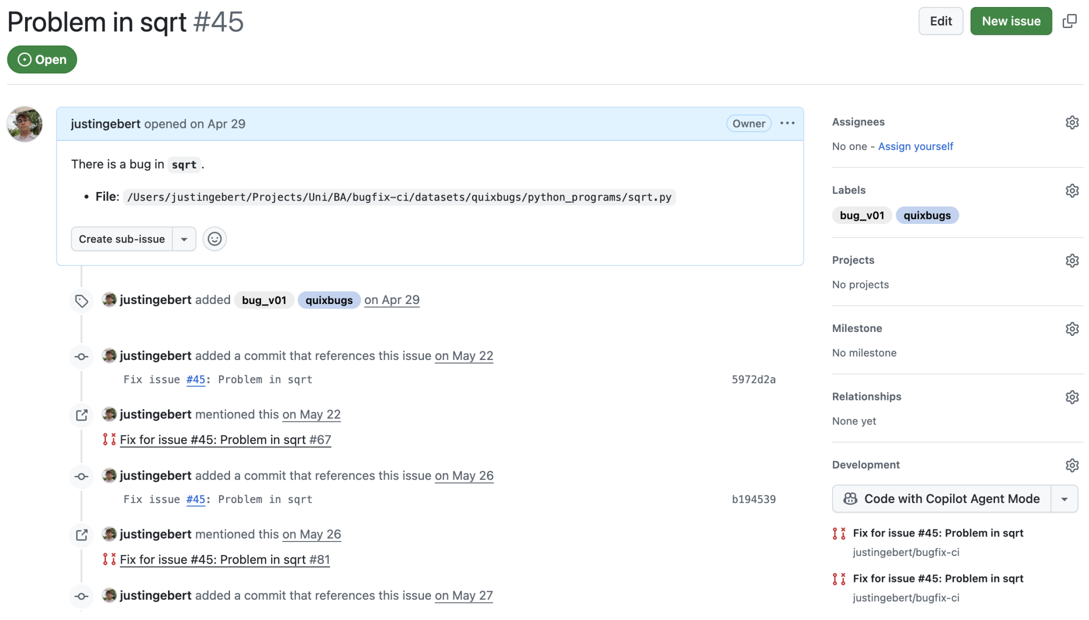
Figure 2.2.: *Continuous Integration Cycle*

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. [**abrahamssonAgileSoftwareDevelopment2017**]

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.


Problem in sqrt #45

Open

justingebert opened on Apr 29

There is a bug in `sqrt`.

- File: `/Users/justingebert/Projects/Uni/BA/bugfix-ci/datasets/quixbugs/python_programs/sqrt.py`

justingebert added bug_v01 quixbugs on Apr 29

justingebert added a commit that references this issue on May 22

```
Fix issue #45: Problem in sqrt                                   5972d2a
```

justingebert mentioned this on May 22

Fix for issue #45: Problem in sqrt #67

justingebert added a commit that references this issue on May 26

```
Fix issue #45: Problem in sqrt                                   b194539
```

justingebert mentioned this on May 26

Fix for issue #45: Problem in sqrt #81

justingebert added a commit that references this issue on May 27

Assignees: No one - Assign yourself

Labels: bug_v01 quixbugs

Projects: No projects

Milestone: No milestone

Relationships: None yet

Development

Fix for issue #45: Problem in sqrt
justingebert/bugfix-ci

Fix for issue #45: Problem in sqrt
justingebert/bugfix-ci

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. [**Workflows**] 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

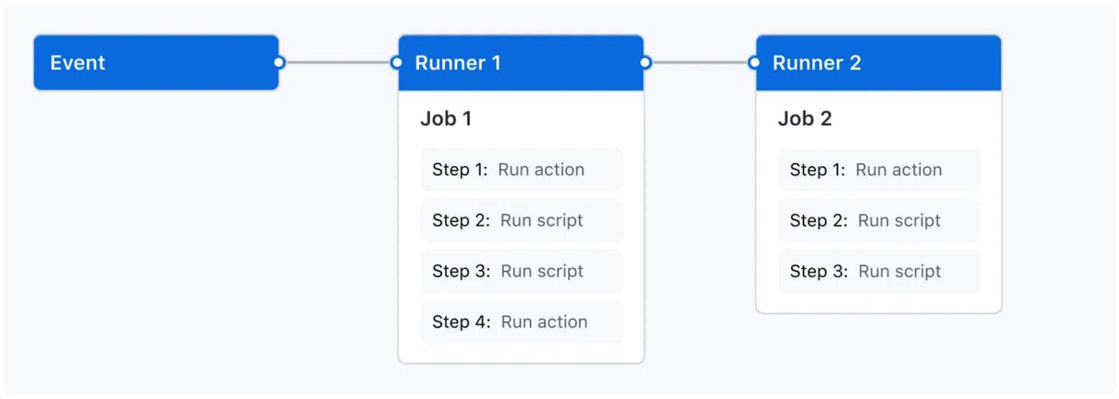


Figure 2.4.: *Simple Action*

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 [**changSurveyEvaluationLarge2024**]. 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 [**LLMsWhatsLarge**]. 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, they can still hallucinate incorrect or biased content. [**LLMsWhatsLarge**].

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. Table 2.1 shows a selection of popular LLMs with their characteristics and performance on NLP benchmark X which evaluates X.

Model Name	Publisher	Parameters	Context Window Size	Cost per 1M
chatgpt	OpenAI	175B	1M	1$

Table 2.1.: *Large Language Model Examples*

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 [**chenUnveilingPitfallsUnderstanding2025**]. 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 [**bhargavmallampatiRoleGenerativeAI2025**]. These tools get applied in various stages of the software development lifecycle, including requirement engineering, code generation, debugging, refactoring, and testing [**houLargeLanguageModels2024**, **puvvadiCodingAgentsComprehensive2025**, **bhargavmallampatiRoleGenerativeAI2025**]. By using LLMs to enhance the named tasks development cycle times can be reduced by up to 30 percent [**bhargavmallampatiRoleGenerativeAI2025**, **kalliamvakouResearchQuantifyingGitHu** Furthermore these tools have positive impacts like improving developer satisfaction and reducing cognitive load [**kalliamvakouResearchQuantifyingGitHub2022**].

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 [**houLargeLanguageModels2024**]. 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 [**bhargavmallampatiRoleGenerativeAI2025**]. When generating code LLMs can produce incorrect or insecure code, which can lead to further bugs and vulnerabilities in the software [**houLargeLanguageModels2024**, **bhargavmallampatiRoleGenerativeAI2025**]. 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 [**liuPromptInjectionAttack2024**]. Code generated by LLMs based on training data also raises questions about ownership, responsibility and intellectual property rights. [**sauvolaFutureSoftwareDevelopment2024**, **houLargeLanguageModels2024**].

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 [**houLargeLanguageModels2024**, **puvvadiCodingAgentsComprehensive2025**]. 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. [**puvvadiCodingAgentsComprehensive2025**, **dohmkeGitHubCopilotMeet2025**, **IntroducingCodex**, **sauvolaFutureSoftwareDevelopment2024**]. This happens in tools and on platform where development happens and focuses on for example integrating AI/ML into CI/CD [**mohammedAIDrivenContinuousIntegration2024**] 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. [**zhangSurveyLearningbasedAutomated2024**] 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. [**houLargeLanguageModels2024**]

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 [**zhangSurveyLearningbasedAutomated2024**, **baderGetafixLearningFix2019**]. 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 [**baderGetafixLearningFix2019**].

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.[**puvvadiCodingAgentsComp**
houLargeLanguageModels2024]

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. [**liuMarsCodeAgentAInative2024**, **xiaAutomatedProgramRepair2023**, **yinThinkRepairSelfDirecte**
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. [**liuMarsCodeAgentAInative2024**, **huCanGPTO1Kill2024**, **zhangPATCHEmpoweringLarge** 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 [**legouesGenProgGenericMethod2012**].
- **Semantics / constraint based repair** synthesizes patches using constraint solvers to based on semantic information of the program and tests. [**liuMarsCodeAgentAInative2024**, **mechtaevAngelixScalableMultiline2016**] Angelix being a prominent example. [**mechtaevAngelixScalableMultiline2016**].
- **Template based repair** relies on mined templates for transformations of known bugs. [**xiaAutomatedProgramRepair2023**] Templates are mined from previous human produced bug fixes.[**xiaAutomatedProgramRepair2023**, **yinThinkRepairSelfDirectedAuton** Getafix being an example of an industrially deployed tool learning recurring fix pattern form past fixes [**baderGetafixLearningFix2019**]

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. [**puvvadiCodingAgentsComprehensive2** **xiaAutomatedProgramRepair2024**]

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 [**xiaAutomatedProgramRepair2023**, **tangLargeLanguageModels2024**]. A prominent of a learning based APR system is CoCoNut [**lutellierCoCoNuTCombiningContextaware2020**], Recoder [**zhuSyntaxguidedEditDecoder2021**]. Despite significant advancements these methods are limited by training data and struggle with unseen bugs. [**xiaLessTrainingMore2022**]

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 [**hossainDeepDiveLarge2024**]. For that reason LLMs lay the groundwork of a new APR paradigm [**chenUnveilingPitfallsUnderstanding2025**, **anandComprehensiveSurveyAIDriven2024**].

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 [**puvvadiCodingAgentsComprehensive2025**]. This approach allows access to external knowledge in the repair process, enhancing the LLM's ability to understand and fix bugs [**houLargeLanguageModels2024**, **yinThinkRepairSelfDirectedAutomated2024**].
- **Interactive/Conversational approaches** make use of LLMs dialogue capabilities to

provide patch validation with instant feedback. [**xiaAutomatedProgramRepair2024**, **huCanGPTO1Kill2024**] This feedback is used to iterate and refine generated patches with the goal of archive better results. [**xiaAutomatedProgramRepair2024**]
- **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. [**anandComprehensiveSurveyAIDriven**
puvvadiCodingAgentsComprehensive2025, **mengEmpiricalStudyLLMbased2024**] Using multi-step reasoning these frameworks reconstruct the cognitive processes of developers using multiple specialized agents. [**rondonEvaluatingAgentbasedProgram2025**, **zhangPATCHEmpoweringLarge2025**, **leeUnifiedDebuggingApproach2024**]. Examples include SWE-Agent [**yangSWEagentAgentComputerInterfaces2024**], Fix-Agent [**leeUnifiedDebuggingApproach2024**], MarsCodeAgent [**liuMarsCodeAgentAInative2024**], 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 [**xiaAgentlessDemystifyingLLMbased2024**, **puvvadiCodingAgentsComprehensive2025**]. 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 [**xiaAgentlessDemystifyingLLMbased2024**, **mengEmpiricalStudyLLMbased2**

Commonly used LLMs for the mentioned APR techniques include ChatGPT, Codex, CodeLlama, DeepSeek-Coder, and CodeT5 [**houLargeLanguageModels2024**, **yinThinkRepairSelfDirecte**
anandComprehensiveSurveyAIDriven2024]. 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.[**xiaAgentlessDemystifyingLLM**
puvvadiCodingAgentsComprehensive2025, **houLargeLanguageModels2024**] The bug fixing process bugs takes a lot of computational resources and is time intensive making the program repair expensive [**sobaniaAnalysisAutomaticBug2023**, **puvvadiCodingAgentsComprehensi**
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[**meemExploringExperiencesAutomated2024**, **puvvadiCodingAgentsComprehensive2**

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. [**anandComprehensiveSurveyAIDriven2024**] They are essential for comparing the performance of different APR systems and understanding their strengths and weaknesses. [**puvvadiCodingAgentsComprehensive2025**] APR benchmarks are available for different programming languages with popular ones being QuixBugs [**linQuixBugsMultilingualProgram2017**], Defects4J [**justDefects4JDatabaseExisting2014**], ManyBugs [**legouesManyBugsIntroClassBenchmarks2015**] and SWE Bench [**jimenezSWEbenchCanLan**
[**wangSoftwareDevelopmentLife2025**]

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 2.2.: *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.

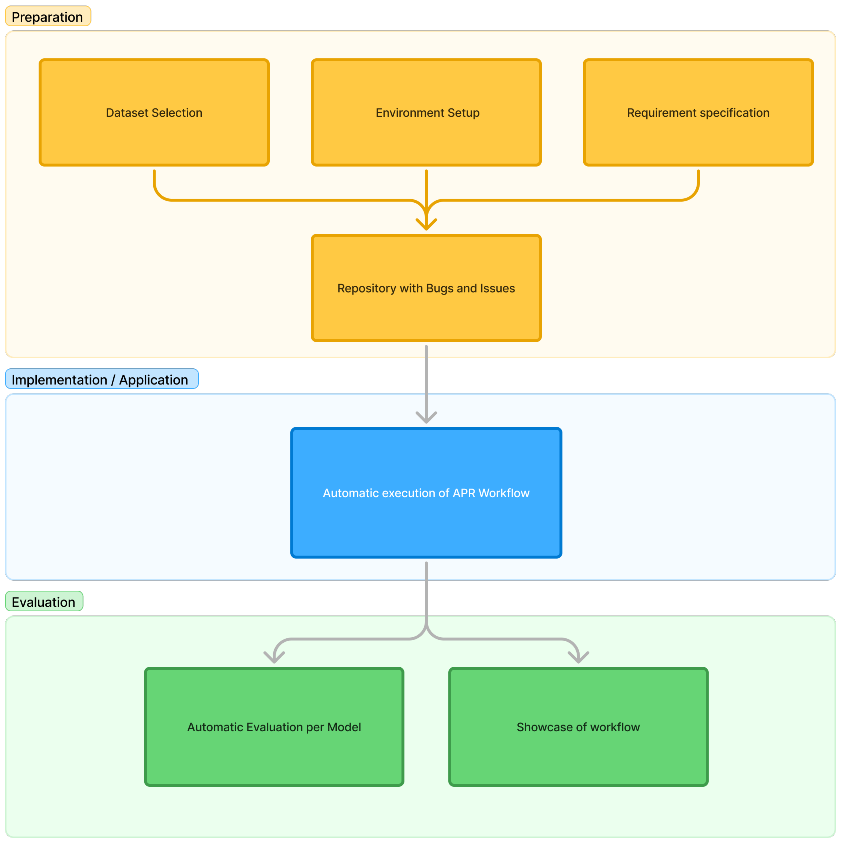

Preparation
Dataset Selection
Environment Setup
Requirement specification
Repository with Bugs and Issues
Implementation / Application
Automatic execution of APR Workflow
Evaluation
Automatic Evaluation per Model
Showcase of workflow

Figure 3.1.: *Thesis Method Overview*

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

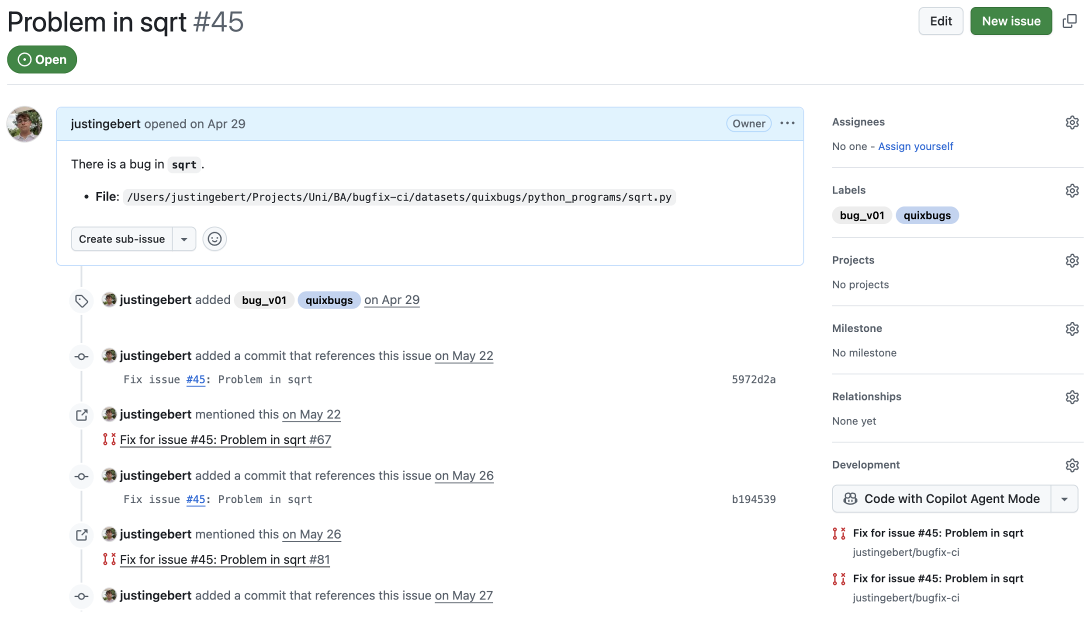
For the evaluation of our APR integration, we selected the QuixBugs benchmark [**linQuixBugsMultilingualProgram2017**]. This dataset is well-suited for our purposes due to its focus on small-scale software bugs in Python. It consists of 40 individual files containing an algorithmic bug each. The bug 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 [**linQuixBugsMultilingualProgram2017**], 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 2.2 like SWE-Bench [**jimenezSWEbenchCanLanguage2024**] QuixBugs is relatively small which allows for accelerated setup and development.

3.1.2. 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.

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.


Problem in sqrt #45

Open

justingebert opened on Apr 29 Owner

There is a bug in `sqrt`.

- File: `/Users/justingebert/Projects/Uni/BA/bugfix-ci/datasets/quixbugs/python_programs/sqrt.py`

justingebert added bug_v01 quixbugs on Apr 29

justingebert added a commit that references this issue on May 22

```
Fix issue #45: Problem in sqrt                5972d2a
```

justingebert mentioned this on May 22

Fix for issue #45: Problem in sqrt #67

justingebert added a commit that references this issue on May 26

```
Fix issue #45: Problem in sqrt                b194539
```

justingebert mentioned this on May 26

Fix for issue #45: Problem in sqrt #81

justingebert added a commit that references this issue on May 27

Assignees
No one - Assign yourself

Labels
bug_v01 quixbugs

Projects
No projects

Milestone
No milestone

Relationships
None yet

Development

Fix for issue #45: Problem in sqrt
justingebert/bugfix-ci

Fix for issue #45: Problem in sqrt
justingebert/bugfix-ci

Figure 3.2.: *Example of a GitHub Issue*

3.1.3. 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 [**10.5555/984017**]. 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:

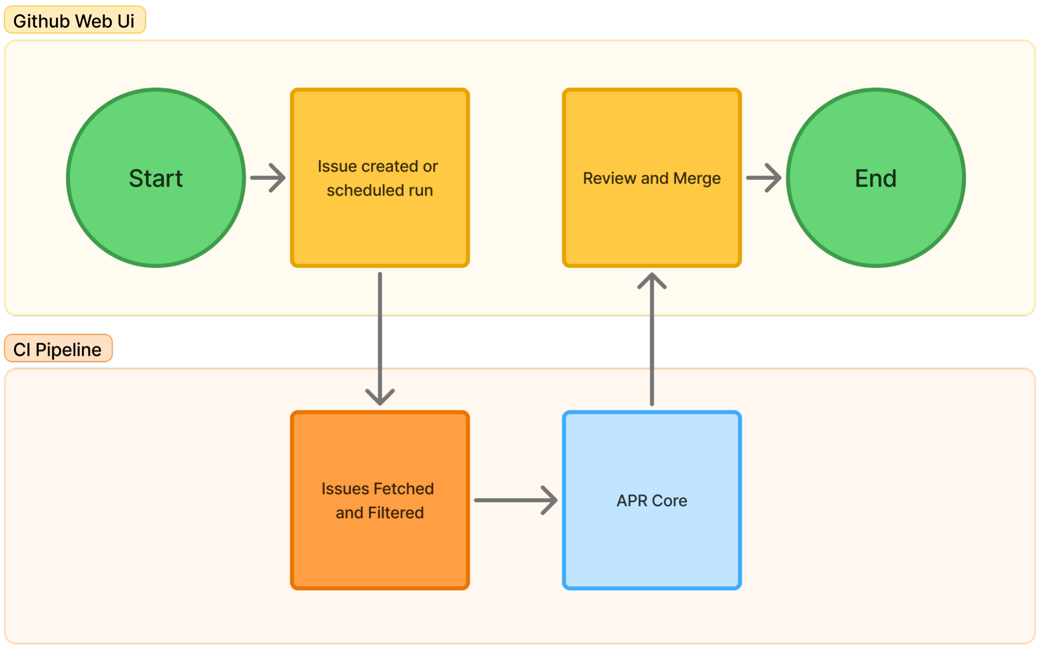


Figure 3.3.: *High-Level Overview of the APR Pipeline*

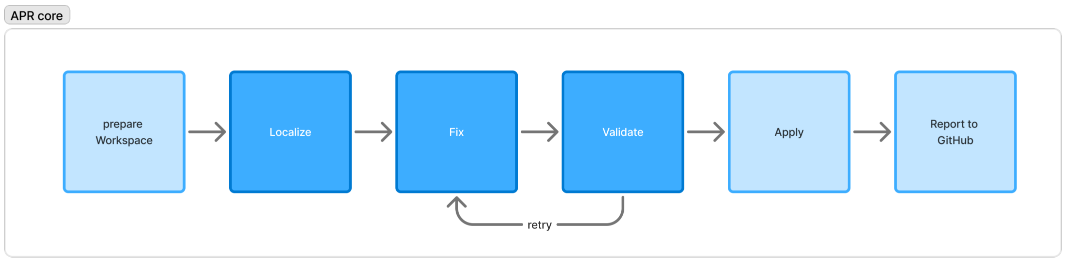


Figure 3.4.: *High-Level Overview of the APR Core*

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 and performance of our APR pipeline when integrated into a Github repository. Using Github Actions Continuous Integration abilities, with our QuixBugs repository as a bases.

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. If both pass the issue is considered repaired.

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 metrics defined in 3.4

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 3.2	APR Core

Table 3.1.: *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	APR Core
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 3.2.: *Run Metrics Collected for each 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 3.3.: *Run Metrics Collected for each stage*

Using this data we calculate the following metrics for each LLM model:

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 3.4.: *Metrics Calculated from the collected data*

4. Requirements

For development we developed the following requirements which guide the implementation and design of the prototype. Following the INVEST model [**10.5555/984017**] 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 (F0–F8)*

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 (N1–N5) 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.

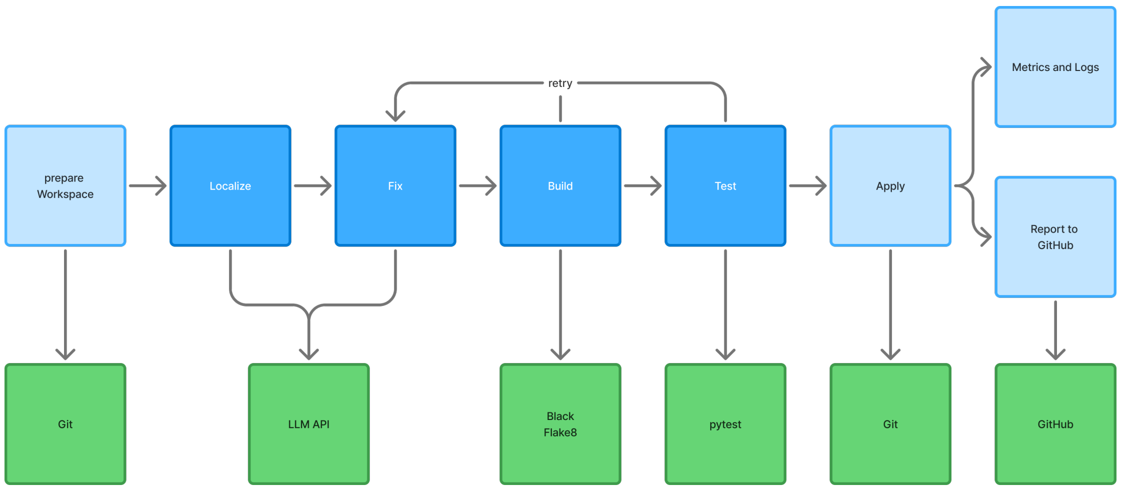


Figure 5.1.: *APR Core Logic*

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.

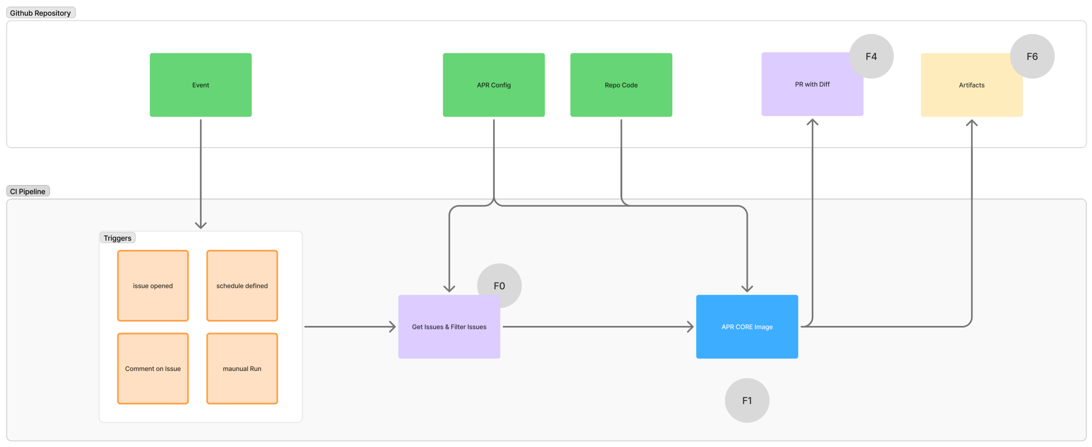


Figure 5.2.: *APR Core Logic*

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

[5]todo

following adjustments need to be made to the repository:

- Add Secrets: LLM provider API key, GITHUB TOKEN
- enable GitHub actions permission to create pull requests in repo settings

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

6. Results

In the following section we will showcase the resulting workflow of our prototype and the evaluation results for our repository containing the QuixBugs benchmark problems.

6.1. Showcase of workflow

To integrate the APR system into a repository living on GitHub we need to move the pipeline with its filter script to the dedicated github action workflow directory. —IMAGE OF WORKFLOW IN PLACE

The follwing repository secrets need to be set in the repository settings GITHUB_TOKEN: a personal access token with write access to the repository LLM_API_KEY: the API key for the LLM provider e.g. OpenAI, Gemini, etc.

When the workflow is in place the APR system is ready to go. Optionally its default behavior can be altered by adding a configuration file (called: bugfix.yml) to the root of the repository. —EXAMPLE OF CONFIG

Now when an issue is created and labeled with the default label "bug" (or a custom label defined in the configuration file) the APR system will be triggered and start the bug fixing process. Manual triggering is also possible by using the "Run workflow" button in the GitHub actions tab of the repository. —IMAGE OF ISSUE BEING CREATED OR MANUALLY TRIGGERED

After the workflow is triggered and relevant issues are found the APR system start as a run of the GitHub action workflow. —IMAGE OF RUN BEING STARTED

When the automatic bug fixing process has completed there are two possible outcomes: Pull Request with patch for bug and link to the issue. —IMAGE OF PULL REQUEST

or a comment on the issue that bug fixing failed after all attempts. —IMAGE OF COMMENT ON ISSUE

After the Workflow finishes metrics and logs are available for download in the action. (during a run logs are live streamed in the Workflow run) —IMAGE OF LOGS

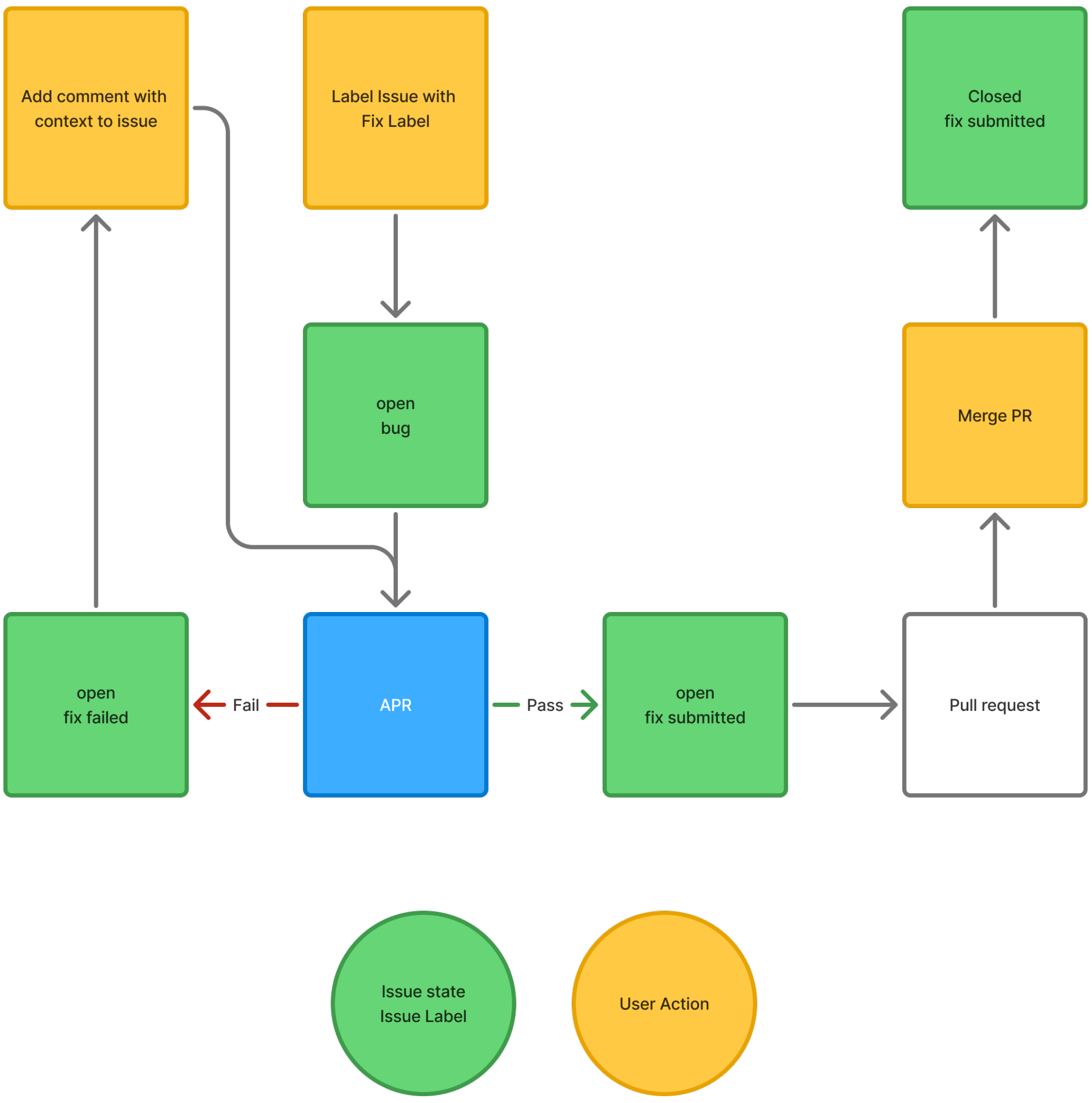


Figure 6.1.: *flow*

6.2. Evaluation Results

Our evaluation is based on the data collected during the execution of the prototype. This information is stored in the artifacts of the Github Actions run. Using the **get_run_data.py** script we collected the APR artifacts and the GitHub Pipeline information using the GitHub API. The script then processes the data and generates a report with the results of the evaluation. With the fetched data we calculate the metrics defined in

"gpt-4o": "input": 2.50, "output": 10.00, "gpt-4.1-mini": "input": 0.40, "output": 1.60, "claude-3-7-sonnet": "input": 3.00, "output": 15.00, "claude-3-haiku": "input": 0.25, "output": 1.25, "claude-sonnet-4-0": "input": 3, "output": 15.00,

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

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

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

7. Discussion

7.1. Validity

- Quixbugs a small dataset, not representative of real world software development - only python, not representative of real world software development - but shows the potential of applying llm based agents in a real world CD/CD environment External QuixBugs only, internal non-deterministic LLM, construct tests is not complete oracle

7.2. Potentials

- can take over small tasks in encapsulated environment without intervention - 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

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

agent architectures produce good results epically paired with containerized environments. [**puvvadiCodingAgentsComprehensive2025**]

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

7.3. Limitations

- github actions from github have a lot of computational noise - workflow runs on every issue and therefor has some ci minute overhead this could be solved by using a github app which replies on webhook events – SECURITY ISSUE: Prompt injection in issue: CI/CD makes this a bit safer?

- its limited to small issues

7.4. Summary of Findings

7.5. Lessons Learned

- ai is a fast moving field with a lot of noise

7.6. Roadmap for Extensions

- 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

8. Conclusion

A. Appendix

A.1. Quell-Code

Github Link:

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