Hochschule für Technik
und Wirtschaft Berlin

University of Applied Sciences

Modular Multi-Stage Agent for Bug Fixing - Analysis of Potentials and Limitations

Abschlussarbeit

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)

an der

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

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

Eingereicht von Justin Gebert [s0583511]

22.07.2025

Danksagung

[Text der Danksagung]

Abstract

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

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

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

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

-add that this approach directly integrates into the development lifecycle reducing configuration ...

Contents

List of Figures

List of Tables

Listings

1. Introduction

Generative AI is rapidly changing the software industry and how software is developed and maintained. The emergence of Large Language Models (LLMs), a subfield of Generative AI, has opened up new opportunities for enhancing and automating various domains of the software development lifecycle. Due to remarkable capabilities in understanding and generating code, LLMs have become valuable tools for developers' everyday tasks such as requirements engineering, code generation, refactoring, and program repair [1, 2].

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

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

However, existing APR approaches are often complex and require significant computational resources [17], making them less 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 [18, 12].

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

- **RQ1:** How can LLM-based automated bug fixing be effectively and efficiently integrated into a CI pipeline?
- **RQ2:** What are the 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 [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].

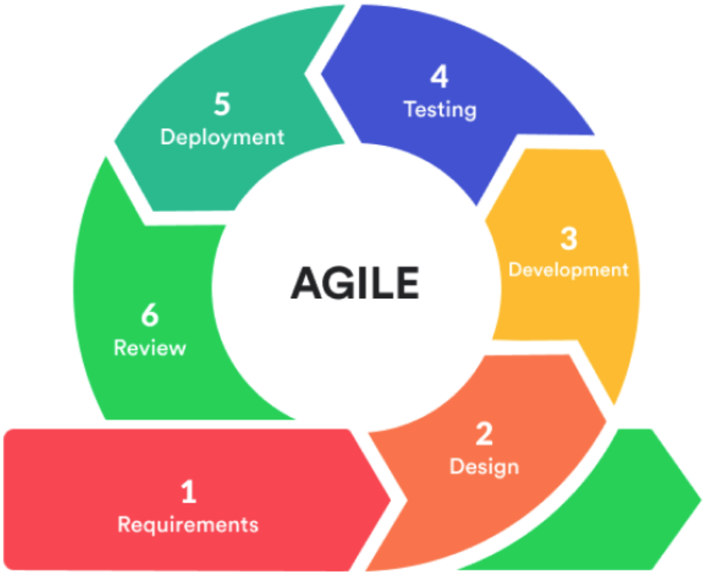


Figure 2.1.: *Agile Software Development Lifecycle*

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].

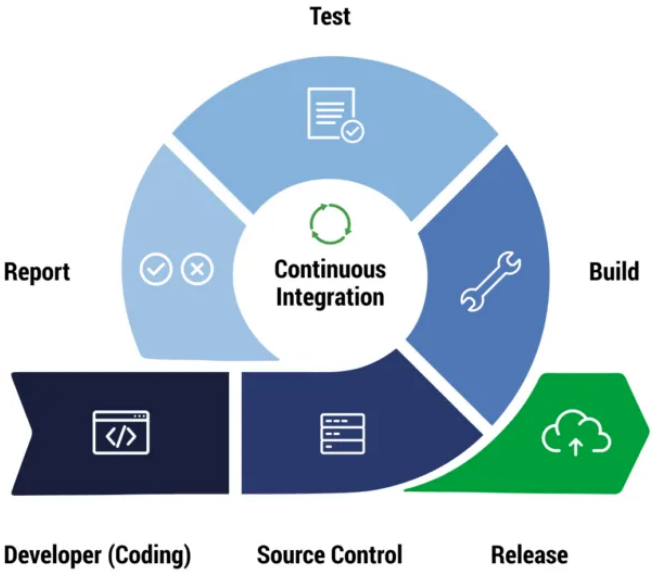


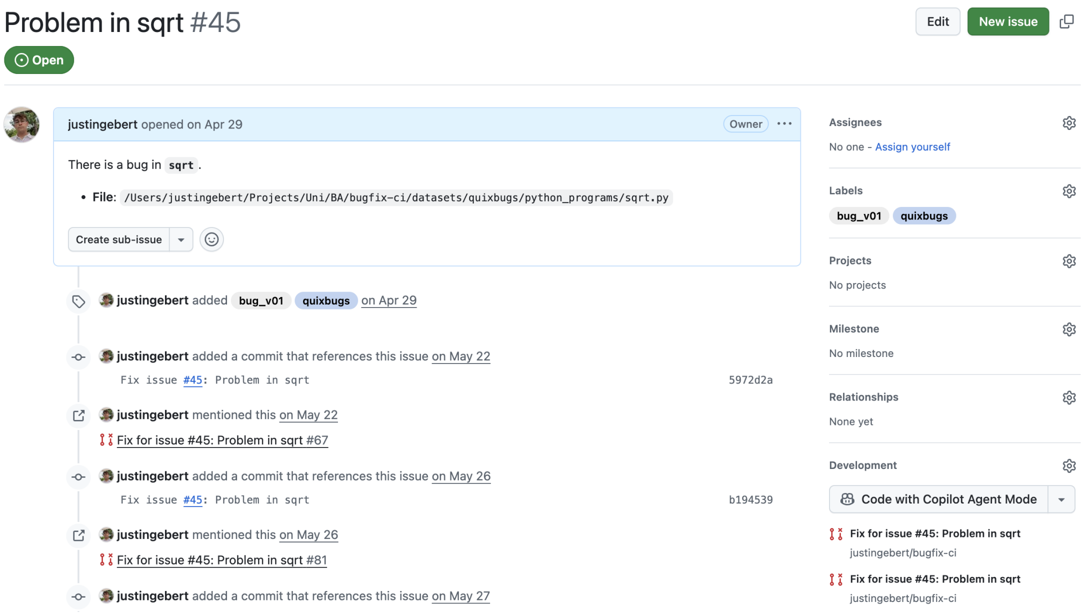
Figure 2.2.: *Continuous Integration Cycle*

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.



Problem in sqrt #45

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. [22] An example is shown in Figure 2.4. Workflow results and logs can be viewed from multiple points in the GitHub web UI , including the Actions tab, the Pull Request page, and the repository's main page. This integration provides a seamless experience for developers to monitor and manage their CI processes directly within their repositories.

[1] TODO explain format

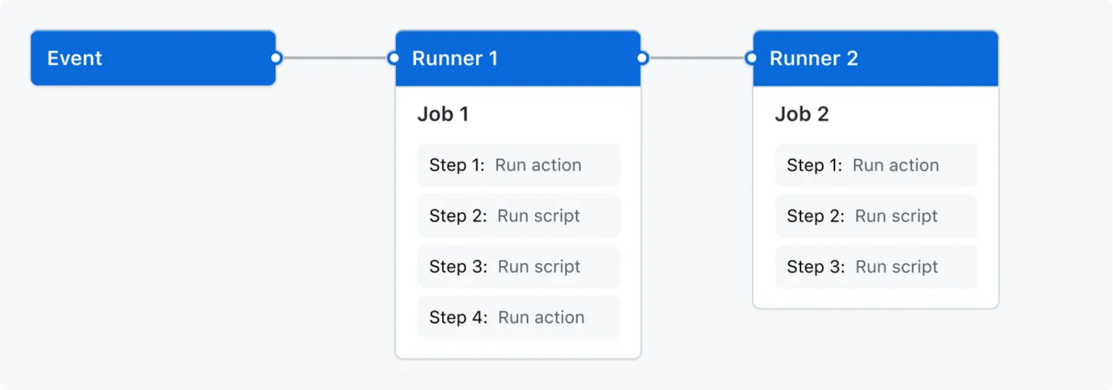


Figure 2.4.: *Simple Action*

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 [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, 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. 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 Windows Size
chatgpt	OpenAI	175B	1M

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 [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 done by software that helps to detect and repair bugs in code with minimal human intervention. This field has also benefited of the rapid advancements in AI.

APR system are supposed to take over the process of fixing bugs therefore making more time for developers to focus on more relevant work. [1]

Specific bugs can be fixed using a resulting patch from an APR system. For creating working patches APR takes a 3 stage approach: First localizing the bug. Then repairing the bug, in the end validation decides where the bug will be passed on.[**<empty citation>**]

In this section we will provide an overview of the evolution of APR, related work, and the current state of APR systems.

2.3.1. Evolution of Automated Program Repair

We have seen multiple transformations 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:
Prior APR approaches were based on version control history, using the history to roll back to a previous version of the code part, where no issues were present. This approach, while effective in some cases, often lacked the ability to preserve new features. (more like instant rollback)

Template based systems relied on predefined template for transformations of commonly known bugs. Templates applied predefined transformations to the code based on fixed rules. This approach had limited the flexibility and adaptability in a quickly transforming software landscape. [2]

Search based repair,

Semantic based repair,

One of the most outstanding system is Getafix develop and deployed at Meta [32]

Nevertheless traditional systems face significant limitations in scalability and adaptability, struggling to generalize to new scenarios or unseen bugs, or to adapt to evolving codebases. They often required extensive computational resources or manual effort. [2]

The emerge of llm based APR: LLM based APR techniques have demonstrated significant improvements over all other state of the art techniques, benefitting from the coding knowledge [33]For that reason LLMs lay the groundwork of a new APR paradigm [18]. Common LLMS used for APR include GPT-4, ChatGPT, Codex, CodeLlama, DeepSeek-Coder, and CodeT5 [1, 34, 35].

Using LLMs diffrent Paradigms have emerged and are being actively researched. These paradigms include:

Retrieval-Augmented Approaches repair bugs with the help of retrieving relevant context during the repair process. This approach allows adding external knowledge to the repair process, enhancing the LLM's ability to understand and fix bugs [1, 34].

Agent based system improve fixing abilites by probiding llms the ability to interact with the code base and the environment, allowing them to plan their actions. These frameworks recosncturct the cognitve processes using multiple Agents that can generate code with the help of multi-step reasoning, usage of tools for with Envrioments and Tools Examples for that are SWE-Agent [13], FixAgent [8], MarsCodeAgent [12], GitHub Copilot.

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

Interactive approaches make use of LLMs dialogue capabilities to equip patch validation with instant feedback. This feedback is used to refine the generated patches trying to archive better results. This process takes a more dynamic and iterative approach. [15]

Agentless systems are recent a push towards more lightweight solution, focusing on simplicity and efficiency. These approaches aim to reduce the complexity of APR systems while maintaining effectiveness in bug fixing [9]. Furthermore this approach provides clear rails to the LLMS improving the transparency of the bug fixing approach taken. These Systems have achieved promising results with low costs [9]

Common problems currently faced by state of the art APR system are: Exsiting system are overly complex with limited transparency and control over the bug fixing process.[**puvvadiCodingAgentsComprehensive202**, 9, 1] Repairing bugs takes a lot of computational resources and is time intensive therefore producing significant costs [14, 2] Repairing bugs is done on benchmarks or in controlled set up environments and not integrated into real world software development workflows [36, 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, which can be used to evaluate the effectiveness of different APR techniques. They are essential for comparing the performance of different APR systems and understanding their strengths and weaknesses. Widely used benchmarks are QuixBugs, Defects4J and SWE Bench. — Table

—TABLE of benchmarks, num of issues, languages, size, complexity, etc.

Model	**Languages**	**Num of Bugs**	**Description**	**Difficulty**
Quixbugs	Python, Java	40	small single line bugs in Python code	Easy
Defects4J	Java	854	real-world Java bugs	Medium
SWE Bench	Python, JavaScript	2294	Real GitHub defects in software engineering repositories	Hard

Table 2.2.: *Functional requirements (F0–F8)*

3. Method

The primary objective of this thesis is to assess the potentials and limitations of our APR pipeline when integrated into a real-world software development lifecycle. We aim to answer the following research question to evaluate the system's capabilities and impact on the software development process:
What are the potentials and limitations of integrating an LLM-based automated bug-fixing pipeline into a CI/CD workflow, as measured by repair success rate, execution times, and cost, and how does this integration impact the overall software development lifecycle?

For answering this question we streamlined this process into three phases Preparation, Implementation / Application and Evaluation.

3.1. Preparation

For implementing and evaluating our system we first need to prepare an environment where the system can be applied. 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 [37]. 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. A bug is always a single erroneous line in a file. For every file there is a corresponding tests for repair validation. Since these bugs where developed as challenging problems for developers [37],it enables us to evaluate if our system can take over the complex fixing of small bugs without developer intervention to prevent context switching for developers.

Compared to other APR benchmarks like SWE-Bench [38] QuixBugs is relatively small which accelerates setup and development.

if achieved I will ad swe bench lite later [38]

3.1.2. Environment Setup

To mirror realistic software development environment, we prepared a GitHub repository containing the QuixBugs dataset. 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 generated 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 to our APR pipeline. —IMAGE of ISSUE

3.1.3. Requirements Specification

Before implementation we constructed functional and non functional requirements to measure process and document the process further. The requirements are detailed in **??**.

3.2. Pipeline Implementation

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

—IMAGE of HIGH LEVEL System Architecture here

When the system is in place in a target repository. An issue with the default or configured bug label can be created. This trigger trigger will spin up a github action runner which executes the APR pipeline. This Pipeline takes to the configured LLM Api (google and openai) to localize and fix the bugy files. With the supplied file edits the code is validated and tested. When validation passes the a pull request with the file changes is automatically opened on the repository, linking the issue and providing details about the repair process. In case of a unsuccessful repair 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 real-world CI process, using our QuixBugs repository as a bases.

For Evaluation we will focus on several key 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. The following metrics are automatically collected and calculated for each run of the APR pipeline:

- **Repair Success Rate:** Calculate the percentage of successfully repaired bugs out of the total number of bugs attempted by using test results. A successful repair is defined as a bug passing all tests associated with it.

- **Number of Attempts:** Track the number of attempts made by the system to repair each bug with a maximum of 3 attempts.
- **Overall Execution Time in CI/CD:** Evaluate the time taken for the system to execute within a CI/CD pipeline, providing insights into its performance in real-world development environments.
- **Execution Times of Dockerized Agent:** Measure the time taken by the Containerized agent and its stages to execute the repair process and the execution times of individual stages.
  With this CICD overhead can be calculated and bottlenecks can be identified
- **Token Usage**: Monitor the number of tokens used by the LLM during the repair process, which can help understanding the cost of repairing and issue and the relation between token usage and repair success.
- **Cost per Issue:** Calculate the cost associated with repairing each bug, considering factors such as resource usage, execution time, and any additional overhead.

explain how these metrics are collected and calculated, including any tools or scripts used to automate the process. explain how repair success rate is determined

ask zhang wether i need to include the evaluation of swe bench lite from the paper or if a ref is enough

explain statistical methods used to analyze the collected data, such as averages, medians, and standard deviations. This will help in understanding the variability and reliability of the results.

explain how resuLTS are calculated for model comparison

what I evaluate: script execution time + CICD overhead one issue vs multiple issue times model vs model metrics costs attempts vs no attempts in all these categories

4. Requirements

- for this prototype we constructed Requirements which the system shall satisfy - we split into functional requirements: t.3.1 (EXPLAINATION), nonfunctional Requirements combined with security requirements

- these requirements allow for better planning and prioritization during development. - the satisfaction of all the requirements will allow for evaluation of the integration into the software development lifecycle

4.1. Functional Requirements

4.2. Non-Functional Requirements

ID	Title	Description	Verification
F0	Multi Trigger	The Pipeline can be triggered: manually, scheduled via cron, or by GitHub 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 F1, 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 show build and test results.
F6	Iterative Patch Generation (retry logic)	If F4 reports failures, retry F4–F5 up to X times.	After retries, either F4 passes or fails with no further retries.
F7	Apply Patch	Commit LLM-generated edits and generate patch.	Git history shows a new commit which is referencing the Github issue
F8	Result Reporting	Open a PR or post a comment on GitHub with the diff and summary metrics.	A PR or comment 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.

Table 4.1.: *Functional requirements (F0–F8)*

ID	Title	Description	Verification
N1	Containerized Execution	All agent code runs in CI runner in a Docker container to isloate it.	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 deployed 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	Oberability	The system provides logs and metrics for each run, including execution times, token usage and success rate.	Logs and metrics files are generated after each run, containing all relevant data.

Table 4.2.: *Non-Functional (N1–N5) requirements*

5. Implementation

Here 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 **??** 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.

System Overview:

The system Consists of two main components: The APR core which hold the core logic for the repair process

The CI/CD Pipeline which put everything together and integrates the github entrypoint of the target repository with the core logic of the agent.

Configuration Layer the programs behavior can be altered by adjusting a configuration. A default YAML configuration is in place which allows for controlling: - labels - workdir - branches - Attempts - LLM models Additionally these is the possibility to overwrite these values for a single repository using a dedicated 'bugfix.yml' configuration which needs to be placed at the root of the repository.

5.1. System Components

Agent Core:
The agent core is written in python and dockerized so its slim and portable. the agent core contains the main bug fixing logic. To start fixing bugs it needs the following environment: The repo where to fix files on - provided using docker volume mount the following Environment Variables: - Github Token - Github Repo - LLM API key - ISSUE TO PROCESS - Github repository

With this environment set the system loops over all issues which are fetched from the environment variables.

For each issue the main APR logic is executed. This main logic consists of tools and stages: —IMAGE here First the workspace (a new branch based on configuration naming) and a repair context is set up. the context is hing of the program its needed at every step and works as the main data structure for the APR system like memory. —JSON of Context init here: A stage uses the context to perform a specific task in the bug fixing process and returns the context with its added context. The stages are: Localize, Fix, Build, Test

With a prepared workspace the repair process start with the localization stage. This stage construct a prompt for the LLM to localize the bug in the codebase. This prompt makes use of a constructed hierarchy of the repositories file structure and the issue description. The LLM is expected to return a list of files and lines where the bug is located. — PROMPT

The results are stored in the context.

With the the localized files in the context the fix stages constructs a prompt with the file content and the issue description. the llm can return code or no changes needed. — Prompt these edits are then applied to the files in the workspace. The context is updated with the new file content.

To ensure the changes are properly formatted the the build stages formats the code using the black formatter and lints the python code to ensure maintainable code.

Next up the test stage runs tests for the fixes files and attaches these results to the context.

if tests do not pass the system will record a new attempt and start over from the state of the fix stage. Additionally a feedback is generated using the previous attempts code and results from the context which gets added to the fix prompt.

if the validation passes or the maximum number of attempts is reached the system will either report and unsuccessful repair to the issue or continue to the application steps

on successful repair the following steps are executed: the file changes are committed and a diff file is generated. the changes are pushed to the remote repo using the github token

a pull request is opened with a description of the changes and a link to the issue and more details about tests and some metrics like the number of attempts and the tokens used for the repair process.

During execution the core logs its actions, which can be used for debugging and analysis. Furthermore it collects metrics such as the number of attempts, execution time, and token usage, which are essential for evaluating the performance of the APR system. Logs and metrics are saved as .log and json files at the end

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 the constraints of a CI/CD environment.

CI/CD Pipeline:
The Pipeline is written in YAML acoording to the Github CICD standard. **??** We will use runners hosted by Github which takes away the overhead of managing our own runners but comes at the cost of unknown performance and availability It is made up of the Triggers and 3 Jobs: 'gate', 'skipped' and 'bugfix'. The triggers are set up first and will allow the execution of the APR process. Triggers can results in two types of runs: processing of all bug issues in correct state on manual execution request of the

workflow (“workflow_dispatch”) or scheduled execution (“cron”). processing a single issue: when an issue is opened and label with the configured labels (“issue_opened and issue_labeled ”) or when extra information is added or edited on an issue in form of a comment.

The trigger event information gets passed as environment variables to the next job “gate” which is responsible for evaluating if the issue should be processed or skipped. This job 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 the gate outputs issues that should be processed the job “bugfix” is executed. This job checks out the current repositiory and mounts it as a volume to the agent core container. Addionally it sets the necessary environment variables mentioned at **??**. For the agent to work perimmsions are set on the job level to allow the agent 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.

for all of this to run the following environment secrets need to be defined in the repo:

5.2. System Architecture

IMAGE of Figma diagram

5.3. System Configuration

secrets that need to be added: LLM provider API key, GITHUB TOKEN need to enable GitHub actions permission to create pull requests in repo settings

explain fields:

The full implementation is listed in Appendix **??**

6. Results

In the following section we will showcase the resulting workflow of our prototype and the evaluation results for the Quixbugs benchmark.

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

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

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

Model	Languages	Description	Difficulty
Quixbugs	Python, Java	40 small single line bugs in Python code	Easy
Defects4J	Java	real-world Java bugs	Medium
SWE Bench	Python, JavaScript	Real GitHub defects in software engineering repositories	Hard

Table 6.1.: *Functional requirements (F0–F8)*

Single Issue Processing: CI overhead

Multi Issue Processing: CI overhead

Also include attempt loop

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

7.2. Potentials

- can take over small tasks in encapsulated environment without intervention - small models can solve more problems with retrying with feedback - this concept is applicable to other python repositories

?? - 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 limit 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

8. Conclusion

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[22] *About Workflows*. https://docs-internal.github.com/_next/data/9uQSGns-DWbCy3Cy8blUA/en/fre pro-team%40latest/actions/concepts/workflows-and-actions/about-workflows.json?versionId=free-pro-team%40latest&productId=actions&restPage=concepts&restPage=workflows-and-actions&restPage=about-workflows. (Visited on 06/25/2025).

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

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

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

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

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

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

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

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

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

[32] 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).

[33] 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).

[34] 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).

[35] 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).

[36] 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).

[37] 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).

[38] 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).

A. Appendix

A.1. Quell-Code

A.2. Tipps zum Schreiben Ihrer Abschlussarbeit

- Achten Sie auf eine neutrale, fachliche Sprache. Keine „Ich"-Form.
- Zitieren Sie zitierfähige und -würdige Quellen (z.B. wissenschaftliche Artikel und Fachbücher; nach Möglichkeit keine Blogs und keinesfalls Wikipedia[1]).
- Zitieren Sie korrekt und homogen.
- Verwenden Sie keine Fußnoten für die Literaturangaben.
- Recherchieren Sie ausführlich den Stand der Wissenschaft und Technik.
- Achten Sie auf die Qualität der Ausarbeitung (z.B. auf Rechtschreibung).
- Informieren Sie sich ggf. vorab darüber, wie man wissenschaftlich arbeitet bzw. schreibt:
  - Mittels Fachliteratur[2], oder
  - Beim Lernzentrum[3].
- Nutzen Sie LaTeX[4].

[1] Wikipedia selbst empfiehlt, von der Zitation von Wikipedia-Inhalten im akademischen Umfeld Abstand zu nehmen [**wikipedia2019**].

[2] Z.B. [**balzert2011**], [**franck2013**]

[3] Weitere Informationen zum Schreibcoaching finden sich hier: `https://www.htw-berlin.de/studium/lernzentrum/studierende/schreibcoaching/`; letzter Zugriff: 13 VI 19.

[4] Kein Support bei Installation, Nutzung und Anpassung allfälliger LaTeX-Templates!

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