SWE-bench-secret: Automating AI Agent Evaluation for Software Engineering Tasks

by

Godsfavour Kio

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2024

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The rise of large language models (LLMs) has sparked significant interest in their application to software engineering tasks. However, as new and more capable LLMs emerge, existing evaluation benchmarks (such as HumanEval and MBPP) are no longer sufficient for gauging their potential. While benchmarks like SWE-bench and SWE-bench-java provide a foundation for evaluating these models on real-world challenges, publicly available datasets face potential contamination risks, compromising their reliability for assessing generalization.

To address these limitations, we introduce SWE-bench-secret, a private dataset carefully selected to evaluate AI agents on software engineering tasks spanning multiple years, including some originating after the models' training data cutoff. Derived from three popular GitHub repositories, it comprises 457 task instances designed to mirror SWE-bench's structure while maintaining strict data secrecy. Evaluations on a lightweight subset, called SWE-Secret-Lite, reveal significant performance gaps between public and private datasets, highlighting the increased difficulty models face when dealing with tasks that extend beyond familiar patterns found in publicly available data.

Additionally, we provide a secure mechanism that allows researchers to submit their agents for evaluation without exposing the dataset.

Our findings emphasize the need for improved logical reasoning and adaptability in AI agents, particularly when confronted with tasks that lie outside well-known public training data distributions. By introducing a contamination-free evaluation framework and a novel secret benchmark, this work strengthens the foundation for advancing benchmarking methodologies and promoting the development of more versatile, context-aware AI agents.

Acknowledgements

I want to start by sincerely thanking Professor Mei Nagappan, my supervisor, without whom I could not be here. I am indebted to him for his invaluable mentorship, patience, kindness, support, and direction. I learned considerably more in these months than I ever imagined thanks to his ideas, leadership and patience with me as I figured out this new area of research. Being a student of his is an honor of a lifetime.

Professors Daniel Berry, Edith Law, Jimmy Lin, and Chengnian Sun are also very much appreciated for the wealth of knowledge they instilled in me while I was a student at the University of Waterloo. They have improved my research abilities and widened my understanding of AI and software engineering. Along with Professo Edith Law, I would like to thank Professor Nancy Day for serving as my thesis readers. Their constructive feedback and thoughtful comments have contributed profoundly to the quality of this work.

I am grateful to the super talented and creative members of the Software Analytics Group (SWAG) and the Software Repository Excavation and Build Engineering Labs (The Software REBELs) whose companionship has been a continual source of inspiration throughout this project. A special mention to Joy, Shaquille, Debbie, Amirezza, Gareema, Akin, Anubhav, Arian, Liz, Noble, Yiwen, Mattie, Nimmi, Albert, Farshad, and Gen. I appreciate the wonderful talks and coffee breaks.

To my family — Mr Ngo Kio (1957 - 2022), Mrs Gloria Kio, Tammy, Ibelemari, Joshua, Mbabi, Treasure, and Onari — you all are my rock; without you and the sacrifices you have made for me, this would not have been possible. I am eternally grateful to have you in my life.

Finally, I would like to thank my friends Obelem, Emefa, Tunmise, Isaac, Tolu, Tayo, Dolapo, amongst a plethora of others, for providing listening ears to me when I needed people to talk to, and encouranging me when I felt lost. Once more, I wouldn't be here without all of you, this win is for all of us.

As a member of the University of Waterloo, I acknowledge that this work took place on the traditional territory of the Neutral, Anishinaabe and Haudenosaunee peoples.

Dedication

To my beloved Father, Mr Ngo Bethuel Kio (1957-2022)
who was more than a father to me, who was my friend.
My greatest wish is to make you proud, always.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Large language models, or LLMs, have transformed software engineering in recent years by enabling AI agents to solve complex programming jobs with amazing efficiency. This is no secret. AI tools like ChatGPT [14] and Github Copilot [6] have demonstrated tremendous promise in solving practical problems and expediting the development process, from automating code completion to fixing software bugs. However, the reliability of the evaluation scores of these models is still an important issue because public benchmarks can sometimes be contaminated [18], making it difficult to determine the actual generalization capabilities of LLMs.

Traditional ways of using LLMs involved giving them direct prompts and getting direct answers. In recent times, Retrieval Augmented Generation (RAG) [10] emerged as a way to get more contextualized, accurate and up-to-date information from LLMs by accessing an external datastore. The system gets the prompt, queries the datastore for relevant information regarding that prompt, and then feeds both the prompt and the information to the LLM in order to have it provide the most up-to-date answer.

Beyond RAG, AI agents[9] have emerged as systems that go beyond the traditional ways of using LLMs as well as newer approaches like RAG[10]. An AI agent in this context, is an intelligent system capable of reasoning and acting autonomously with little human intervention. These AI agents are able to perceive their environments, being able to go into a codebase and not just generate code, but add or modify files in different folders of the codebase, just like a human Software Engineer would.

For assessing AI agents on repository-level tasks, benchmarks like SWE-bench [12] and SWE-bench-java [23] have become well-known. Through the use of tasks based on pull requests and GitHub issues, these benchmarks highlight useful problem-solving in real-life

situations. Notwithstanding their advantages, the risk of contamination is introduced by the public availability of these datasets, since models could unintentionally come across training data that overlaps with benchmark tasks, jeopardizing the validity of evaluation outcomes.

In order to alleviate these worries, we introduce SWE-bench-secret, a private dataset that has been carefully selected to evaluate AI agents on real-world software engineering tasks. The key difference being that researchers do not have access to the dataset at any point before or after the evaluation of their agents. SWE-bench-secret, which consists of 457 tasks from 3 popular repositories, replicates structure of SWE-bench. This approach provides an evaluation environment devoid of contamination, which offers a reliable benchmark for evaluating AI agents' generalization abilities in situations that mimic real-life situations.

This work presents an evaluation infrastructure that can be used for evaluating agents on SWE-bench-secret, in addition to the dataset. Through containerized environments, the infrastructure automates agent submission, evaluation, and artifact generation. The integrity of the evaluation process is protected by anonymizing sensitive data before returning them back to researchers.

This infrastructure gives researchers actionable information through comprehensive data on agent performance, while also doing away with the manual evaluation and submission process that is characteristic of SWE-bench. Despite being anonymized, we do not claim to make it impossible for researchers to determine what repositories the tasks are from. However, by making it as "secretive" as possible, we make it harder for them.

This thesis also introduces SWE-secret-lite, a lightweight version of SWE-bench-secret designed for preliminary tests. The dataset has been used to evaluate two AI agents: SWE-Agent and Autocoderover. Comparing our results to publicly available benchmarks reveals significant performance differences, emphasizing the challenges agents face while executing hidden tasks.

The contributions of this study fortify the foundation for advancing AI capabilities in software engineering by resolving the potential drawbacks of existing benchmarks and opening the door for more potent and contamination-free evaluation methods. The SWE-bench-secret ecosystem's ongoing reliability, versatility, and researcher-friendliness are further supported by the implementation of a secure infrastructure, setting a new standard for contamination-free software engineering benchmarking.

1.1 Contributions

Figure 1.1 provides an overview of the scope of this thesis. The following research questions are answered:

1. **RQ1: How can we design a contamination-free dataset for evaluating AI agents on real-world software engineering tasks?**
   To address this, we introduce **SWE-bench-secret**, a secret dataset comprising 457 tasks sourced from three popular GitHub repositories. By secret, we mean that researchers do not have access to the dataset. Their agents are submitted to us, and we evaluate them under controlled environments, before returning the results of the evaluation. At no point before or after the evaluation process do researchers have access to the data. The dataset mirrors the structure of SWE-bench, which means that agents that have already been trained on SWE-bench do not have to be trained again, they can undergo an evaluation that can be as contamination-free as possible because they would not have had any access to the testing set like SWE-bench has publicly available.

2. **RQ2: What infrastructure is needed to ensure secure evaluation of AI agents?**
   To answer this, we develop a **secure evaluation infrastructure** utilizing containerized environments. This infrastructure automates agent submission, evaluation, and artifact generation while maintaining data confidentiality. The platform aims to simplify the evaluation and submission process for researchers. We make publicly available our evaluation infrastructure for researchers and practioners to test their agents on SWE-bench-secret.

3. **RQ3: What are the strengths and limitations of current state-of-the-art AI agents in addressing tasks from varying time periods, including those likely unseen by the models?**
   We evaluate two leading open-source AI agents, **SWE-Agent** and **Autocoderover**, on the SWE-secret-lite dataset, which includes tasks spanning multiple years. Notably, some tasks originate from 2024, a period beyond the models' training data cutoff and thus more likely to be truly unseen, while others stem from earlier years that may overlap with publicly available data. Our analysis reveals performance differences across these temporal segments, highlighting both the agents' potential to handle novel scenarios and the challenges they face when confronted with tasks that do not align with their training data distribution.

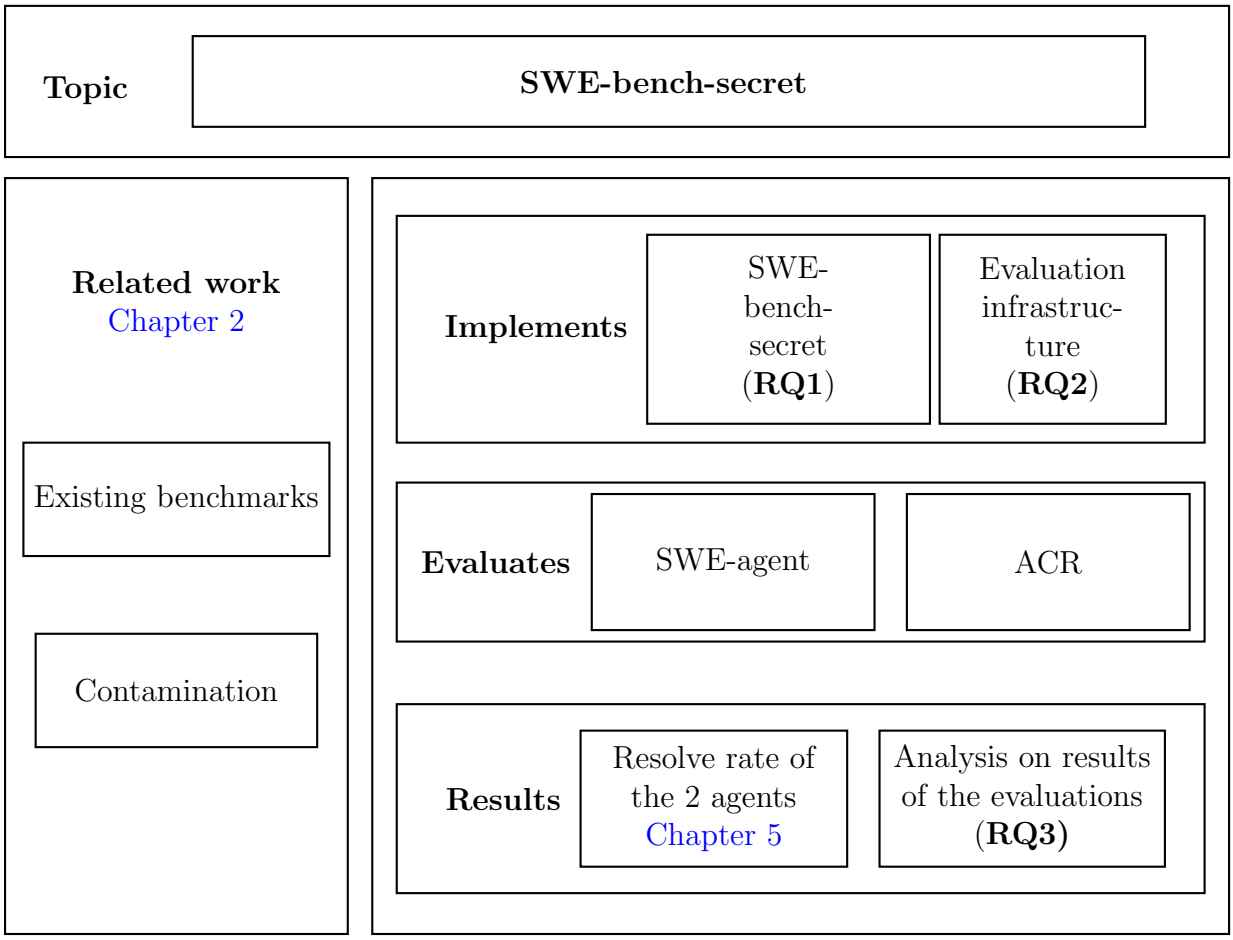

Figure 1.1: An overview of the scope of this thesis.

The rest of this thesis is organized as follows: Chapter 2 discusses related work. Chapter 3 describes the approach used in curating our dataset. Chapter 4 presents the infrastructure developed to make secure evaluations on SWE-bench-secret possible. In Chapter 5, we discuss the results and provide some further analysis. Chapter 6 highlights the threats to validity. Chapter 7 concludes the paper and discusses future work.

Chapter 2

Related Work

The advent of large language models (LLMs) has revolutionized software engineering, sparking interest in their evaluation through diverse benchmarks. This chapter reviews key developments in this domain, focusing on the evolution of benchmarks, advancements in machine learning for software engineering, and the challenges inherent in creating realistic evaluation frameworks.

2.1 Benchmarks for Automated Software Engineering

Benchmarks play a crucial role in evaluating LLMs' abilities to understand, generate, and refine code. Early efforts, such as HumanEval, introduced a collection of Python programming problems to assess functional correctness in code generation [4]. Similarly, MBPP evaluates LLMs on basic Python programming skills using a set of diverse yet simple programming tasks [2]. While effective for small, self-contained problems, these benchmarks often lack the complexity needed to reflect real-world software engineering scenarios. The single-function problems in HumanEval do not account for the fact that developers frequently work with many files and existing codebases in real-world software engineering.

SWE-bench addresses these limitations by introducing tasks derived from real-world GitHub issues and pull requests, requiring LLMs to resolve repository-level challenges. It curates these tasks from 12 popular repositories. Its focus on complex, multi-file problems sets it apart from other benchmarks. SWE-bench Lite, a simplified variant, offers computationally lighter evaluations while retaining task diversity [12]. Building on this

foundation, SWE-bench-java extends the framework to Java repositories, enabling multilingual evaluations and addressing the monolingual limitations of its predecessors [23].

In addition to repository-level benchmarks, library-specific benchmarks like PandasEval and NumpyEval assess LLMs' proficiency in generating code that interacts with popular Python libraries [22, 8]. RepoBench and CrossCodeEval take this further by challenging models with real-world, cross-file programming tasks, emphasizing multi-file understanding [13, 5].

2.2 Advancements in Machine Learning for Software Engineering

The application of LLMs in software engineering has expanded significantly. We evolved from giving simple prompts to LLMs to approaches like Retrieval Augmented Generation[10], and we have arrived at the agentic approach to using LLMS using AI agents[9]. In the context of code generation particularly with regards to SWE-bench, agent-based frameworks, such as SWE-Agent, demonstrate the potential of LLMs to autonomously navigate repositories, edit code, and validate patches through iterative execution [21]. Similarly, frameworks like AutoCodeRover leverage multi-agent systems to refine and optimize codebases, showcasing the adaptability of LLMs in program improvement tasks [24]. Other notable agents include OpenHands's CodeAct v2.1 variant [19], which currently ranks 3rd on the SWE-bench-lite leaderboard with a resolve rate of 41.67%. Agentless-1.5 [20] holds 6th place with a resolve rate of 40. 67% at the time of writing this thesis.

Multilingual benchmarks have emerged to address the global nature of software development. For example, MultiPL-E extends HumanEval and MBPP to 18 additional programming languages, providing a comprehensive evaluation across diverse programming paradigms [3]. This shift toward multilingualism demonstrates the growing demand for LLMs capable of handling diverse coding styles and languages.

2.3 Contamination-Free Benchmarks

In simple terms, a model is considered contaminated if it has been evaluated on training instances or trained on validation or test cases. Memorization is a related idea. Data contamination and test-case leakage have been widely discussed in the context of large language models (LLMs), as these models may inadvertently be trained on evaluation

benchmarks. Oren et al. [16], Golchin et al. [7], Anwar et al. [1], Zhou et al. [25] and Roberts et al. [17] have emphasized the risks of contamination, which can compromise the reliability of evaluation results. Sainz et al. [18] showed that some popular benchmark datasets are already memorized by ChatGPT. They demonstrated contamination by simply prompting the model to identify instances of contamination in its training data.

One of the key challenges in evaluating LLMs is avoiding data contamination, where evaluation tasks inadvertently appear in training data. LiveCodeBench addresses this issue through live updates and careful curation. It continuously incorporates new problems from platforms like LeetCode, AtCoder, and CodeForces, ensuring fair evaluations by segmenting problems based on their release dates relative to model cutoff dates [11]. This contamination-free strategy is pivotal in preventing overfitting and maintaining benchmark reliability.

While our approach shares the goal of evaluating LLMs on diverse software engineering tasks, it diverges in methodology. Unlike LiveCodeBench, which relies on competition problems and test-based scenarios like code generation and self-repair, our work emphasizes real-world repository-level challenges, derived from GitHub issues and pull requests. This distinction allows us to focus on practical, applied problem-solving in real-world contexts, providing a complementary perspective to LiveCodeBench's focus on competition-level coding tasks.

Chapter 3

SWE-bench-secret

3.1 Overview

A major contribution of this research is the curation of a secret dataset that evaluates AI agents on their ability to resolve real-world software engineering issues. The motivation for creating a secret dataset stems from the potential issue of contamination. Existing datasets, like the publicly available SWE-bench, can be used for training agents and models, allowing them to optimize their performance on known tasks. This creates a risk of overfitting to the benchmark and reduces the ability to assess true generalization.

Our secret dataset addresses this by presenting a new benchmark designed to mirror SWE-bench in structure and properties [12], while remaining completely hidden from the public domain. The key question is: what is the relative performance of these AI agents when they encounter a dataset they have clearly never been trained on? Our dataset maintains the same structure as SWE-bench to ensure compatibility with existing AI agents, enabling evaluations without requiring modifications to the agents' core logic. This chapter details the construction, characteristics, and challenges of the SWE-bench-Secret dataset.

3.2 Dataset Construction

We construct our benchmark from three popular repositories, resulting in a final collection of 457 task instances. Each task mirrors the structure of a task instance in the SWE-bench dataset, ensuring consistency and compatibility. This section outlines the repository selection criteria, task instance extraction, and filtering processes.

3.2.1 Repository Selection Criteria

We used the following criteria while choosing repositories in order to create a solid and varied dataset:

- The repository must have a large number of stars on GitHub (minimum 1,000), reflecting its popularity and community engagement.
- The repository must be well-maintained, implementing comprehensive unit testing to ensure the reliability of AI-generated predictions.
- The repository must have a significant number of commits, making sure a rich history of code changes and issues to mine task instances from.

The repositories, anonymized as **Repo1**, **Repo2**, and **Repo3**, were selected based on the following characteristics, with all values approximated to be at least:

- **Repo1:** At least 37,500 stars, 25,000 commits (2015–2024), and 3,000 collaborators.
- **Repo2:** At least 63,000 stars, 54,000 commits (2012–2024), and 5,000 collaborators.
- **Repo3:** At least 2,000 stars, 10,000 commits (2009–2024), and 300 collaborators.

These repositories were chosen for their active maintenance, widespread use, and collaborative development practices. Comprehensive unit testing ensures reliable validation of predictions, even when AI-generated solutions deviate from the original merged patches. If an AI solution passes the tests, it is still considered correct, regardless of differences from the original patch.

3.2.2 Task Instance Extraction

Task instances are mined from pull requests (PRs) and associated issues using the GitHub API. A task instance is considered valid if it meets the following criteria:

1. **Issue Resolution:** The PR must resolve a documented GitHub issue, providing a clear problem statement and context.
2. **Test Modifications:** The PR must include additions or modifications to test files, emphasizing the importance of validation.

3. **Fail-to-Pass Transition:** Application of the PR's patch must transition at least one test case from failing to passing.

Following Jimenez et al. [12], we collected approximately 80,000 PRs from the three selected repositories. Candidate tasks were filtered based on the above criteria, resulting in an initial pool of 27,931 PRs for Repo1, 50,859 for Repo2, and 1,816 for Repo3. Further refinement using the first two criteria reduced this to 427 tasks for Repo1, 2,075 tasks for Repo2, and 64 tasks for Repo3, totaling 2,566 instances. These were validated by running tests before and after applying the solution, yielding a final dataset of 457 task instances.

3.3 Dataset Statistics

3.3.1 Task Distribution by Repository

The final dataset comprises task instances distributed across the three repositories as follows:

- **Repo1:** 95 tasks
- **Repo2:** 312 tasks
- **Repo3:** 50 tasks

This distribution reflects the varying activity levels and sizes of the repositories, providing a diverse set of challenges for AI agents.

3.3.2 Temporal Distribution

The tasks span a range of years, with some originating as recently as 2024. Figure 3.1 illustrates the distribution of tasks over the years, highlighting periods of heightened development activity. Notably, the dataset includes tasks from 2023 and 2024, providing opportunities to evaluate AI agents on issues that were unlikely to be included in training data for existing models.

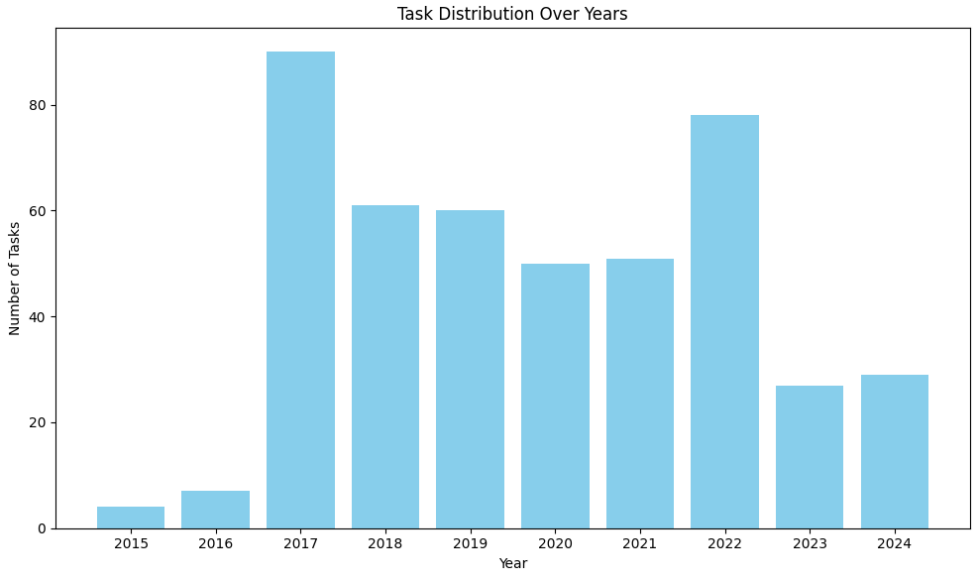

Figure 3.1: Distribution of Tasks Over Years

3.3.3 Task Characteristics

Each task instance encapsulates real-world challenges, requiring solutions that fix the issue, adhere to coding standards, and pass all relevant tests. Every task in the dataset includes the following fields:

- **repo:** The name of the repository.
- **instance_id:** The unique identifier for each task instance.
- **issue_numbers:** A list of every GitHub issue number resolved by the task instance.
- **patch:** The accepted solution merged into the repository (gold patch).
- **test_patch:** Tests added or modified.
- **problem_statement:** The problem statement of the GitHub issue.
- **version:** The repository version at the time the PR was merged.
- **FAIL_TO_PASS:** Tests that transitioned from fail to pass.

- **PASS_TO_PASS:** Tests that remained passing.

Additional fields include `pull_number`, `environment_setup_commit`, and `base_commit`.

3.4 Challenges and Contributions

3.4.1 Addressing Original Guideline Issues

Some challenges from the SWE-bench guidelines were encountered and addressed:

- **Issue-PR Mismatch:** The regex pattern used in identifying PRs that resolve issues was enhanced to handle complex references, including cross-repository issue links and variations in formatting.
- **Validation Efficiency:** True parallel execution was achieved by assigning tasks with the same repository version to individual workers, reducing validation times significantly.

3.4.2 Introduction of Modification Steps

A key contribution of this work is the introduction of **modification steps**, which provide a systematic mechanism to adjust repository files to accommodate specific installation or testing requirements. These steps are particularly useful for resolving issues such as mismatched dependencies, deprecated syntax, or repository-specific patches that cannot be addressed through standard installation commands alone.

One of the key challenges we encountered involved outdated dependency specifications in some of our repositories, particularly in `requirements.txt` files dating back to 2015. In several cases, these files contained a mix of tightly constrained dependencies (e.g., `pytest==4.6.3`) alongside unconstrained ones (e.g., `codecov` without a specified version). This lack of strict versioning led to installation conflicts when resolving dependencies in 2024. Specifically, the package manager attempted to satisfy both the explicitly pinned versions and the latest available versions of unconstrained dependencies, resulting in version mismatches and failed installations.

To mitigate this issue, we introduced `modification_steps`, which enable controlled modifications to dependency files before installation. Each modification step specifies:

- `before_word`: The code snippet or pattern to locate in a given file.
- `after_word`: The replacement code snippet or pattern that should take the place of the matched content.
- `file_path`: The file where this modification should be applied.

For example, consider a scenario where an older version of `pytest` needs to be installed to ensure compatibility with a particular repository version. A `modification_step` can be used to explicitly pin `pytest` to `4.6.3` in the repository's `requirements.txt` file before running tests:

```
"modification_steps": [
    {
        "before_word": "pytest",
        "after_word": "pytest==4.6.3\n",
        "file_path": "requirements.txt"
    }
],
```

By applying these structured modifications, we ensure that dependency resolution remains consistent and reproducible, preventing conflicts that arise due to uncontrolled versioning. This process is essential for adapting older repositories to modern environments while maintaining the integrity of the evaluation framework.

3.5 Regular Future Updates

Our dataset can be continuously updated at set intervals to ensure its relevance in evaluating AI agents against evolving real-world software engineering challenges. The process for collecting the benchmark is largely automated, meaning we can add more task instances without requiring extensive manual intervention.

To facilitate continuous updates, we have an "updating.md" file that provides documentation on extending the dataset beyond the training cutoff dates of state-of-the-art models at any given time.

By maintaining detailed documentation and an automated collection pipeline, future iterations of the benchmark can incorporate additional instances beyond the training data

of contemporary AI models. This ensures that SWE-bench-secret continues to serve as a robust evaluation framework for assessing the generalization capabilities of AI agents in software engineering tasks.

Chapter 4

Infrastructure

4.1 Overview

The infrastructure supporting this research is crafted to evaluate AI agents on the SWE-bench-secret dataset under controlled conditions. We provide a way for researchers to submit their agents to us and have those agents evaluated on our dataset without revealing the dataset to the researchers. This prioritizes security, enabling external agents to participate in evaluations while safeguarding private repository data. This chapter outlines the motivation, architecture, agent submission process, compatibility requirements, and evaluation results, emphasizing how our infrastructure simplifies and enhances the submission and evaluation process.

The decision to select **Autocoderover (ACR)** and **Swe-Agent** was guided by their public availability, open-source nature, and compatibility with Docker. Both agents were listed on the SWE-bench Lite leaderboard, aligning with our goal of working with agents that are already benchmarked and could either build or pull Docker images. These attributes ensured their suitability for integration into our system. Additionally, **GPT-4o** was used as the underlying language model for these agents.

The motivation for building this infrastructure stemmed from the challenges of the original SWE-bench submission process. With SWE-bench, individual AI agent developers must manually evaluate their agents, requiring extensive setup and adherence to strict submission requirements. For example, the SWE-bench [12] website outlines a detailed, multi-step process involving forking, cloning, evaluating, generating artifacts, and creating metadata before submitting a pull request to the leaderboard repository. This process places significant technical and logistical burdens on researchers.

Our system addresses these limitations by automating the entire pipeline. Users simply submit their agents via a structured configuration file, and the system handles everything: evaluation, artifact generation, anonymization, and results compilation. This streamlined approach eliminates manual steps, ensuring researchers can focus on building and refining their agents without navigating complex submission workflows.

4.2 System Architecture

Our infrastructure is built on a pipeline that synchronizes multiple components to evaluate AI agents while preserving data confidentiality. The process begins with Docker image acquisition, where the system either pulls a pre-existing image or builds one from the agent's specified GitHub repository.

Once the container is set up (instantiated from the acquired or newly built Docker image), additional setup commands are executed to configure the environment. These may include dependency installation, virtual environment activation, or runtime-specific adjustments. Input data is provided without direct exposure to the researchers, as the data remains hidden inside the containerized environment. Predictions and logs are written to designated output directories. The system then evaluates predictions using the SWE-bench framework and generates a performance report. Before making the logs available for download, repository names and other identifying details are anonymized using a custom pattern.

The evaluation folder, containing results and anonymized logs, is then zipped and made available for download. Figure 4.1 illustrates a Django patch from the publicly available SWE-bench dataset and Figure 4.1 shows the same patch anonymized as an example. The system architecture (Figure 4.3) and pipeline overview (Figure 4.4) offer a high-level view of this process.

A Flask-based server acts as the central interface for submissions and status tracking, while Celery with Redis manages tasks asynchronously, ensuring that new jobs are queued and processed only after previously running jobs have completed. Leveraging Docker containerization provides isolated environments for each agent. Together, these components form a practical framework that enables researchers to submit their agents for evaluation without directly accessing sensitive data.

```
diff --git a/django/conf/global_settings.py b/django/conf/global_settings.py
--- a/django/conf/global_settings.py
+++ b/django/conf/global_settings.py
@@ -304,7 +304,7 @@ def gettext_noop(s):

 # The numeric mode to set newly-uploaded files to. The value should be a mode
 # you'd pass directly to os.chmod; see https://docs.python.org/library/os.html#files-and-directories.
-FILE_UPLOAD_PERMISSIONS = None
+FILE_UPLOAD_PERMISSIONS = 0o644

 # The numeric mode to assign to newly-created directories, when uploading files.
 # The value should be a mode as you'd pass to os.chmod;
```

Figure 4.1: Example Django patch before being anonymized.

```
diff --git a/repo/conf/global_settings.py b/repo/conf/global_settings.py
--- a/repo/conf/global_settings.py
+++ b/repo/conf/global_settings.py
@@ -304,7 +304,7 @@ def gettext_noop(s):

 # The numeric mode to set newly-uploaded files to. The value should be a mode
 # you'd pass directly to os.chmod; see https://docs.python.org/library/os.html#files-and-directories.
-FILE_UPLOAD_PERMISSIONS = None
+FILE_UPLOAD_PERMISSIONS = 0o644

 # The numeric mode to assign to newly-created directories, when uploading files.
 # The value should be a mode as you'd pass to os.chmod;
```

Figure 4.2: Example Django patch after being anonymized.

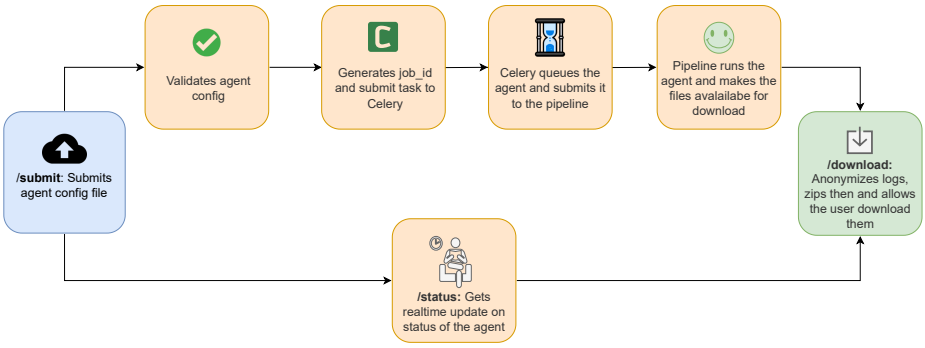

Figure 4.3: High-level architectural overview of the infrastructure.

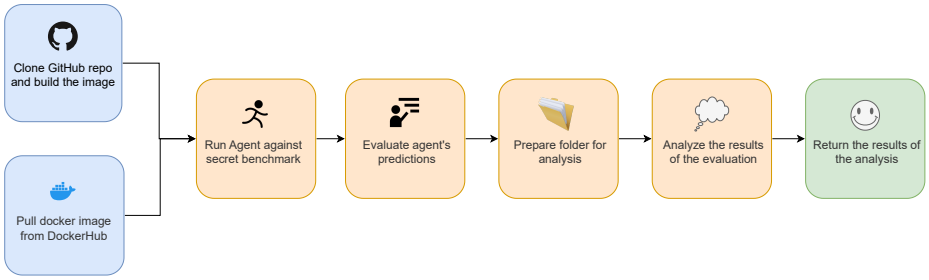

Figure 4.4: High-level overview of the pipeline.

4.3 Agent Submission

Submitting an AI agent to the system requires a structured `agent_config` JSON file. This file defines the agent's runtime requirements, including the repository URL, Docker image (if pre-built), pre-run setup commands, and the main execution command. It also specifies environment variables, volume mappings for input and output data, and other runtime dependencies.

The submission process is automated as follows:

1. The `agent_config` file is validated to ensure all required fields are present and correctly formatted.
2. A unique job ID is generated, and the job is queued for processing.
3. Job status can be tracked in real time via an API endpoint.
4. Once the evaluation is complete, anonymized results, including evaluation reports and logs, are made available for download via another API endpoint.

This streamlined process simplifies integration for researchers while maintaining the integrity of the evaluation pipeline. Table 4.1 details the keys used in the `agent_config` file at the time of writing.

Table 4.1: Description of each field in the agent configuration JSON object.

Field	Description
`github_url` (str)	URL of the GitHub repository where the agent is hosted. If empty, it implies no direct repository link is provided.
`agent_name` (str)	Name of the agent submitting the task. Acts as an identifier for the specific AI agent.
`model_name` (str)	Specifies the model name or version (e.g., `azure/gpt-4o-6824`).
`split` (str)	Indicates the data split being used for evaluation (e.g., `secret` or `lite`).
`date` (str)	The submission date of the agent configuration, typically in YYYYMMDD format.
`run_id` (str)	Unique identifier for the run, used for tracking execution and evaluation.
`per_instance_cost_limit` (float)	Maximum cost allowed per instance during evaluation. Used for limiting the compute or API usage cost.
`docker_image` (str)	Specifies the Docker image to use for the agent. Can be pre-built or constructed during the pipeline.
`pre_run_commands` (list)	Shell commands executed before the main process begins (e.g., `docker build`, package installations).
`docker_container_commands` (list)	Commands executed inside the container, such as creating files or activating environments.
`run_command` (str)	Primary execution command for running the agent. Placeholders like {`model_name`} are interpolated at runtime.
`predictions_output_message` (str)	Message printed upon successful generation of predictions.
`env_vars` (dict)	Environment variables used during execution, such as API keys and configuration paths.
`volume_mappings` (list)	Defines host-to-container volume mappings for files or directories required at runtime.

4.4 Agent Compatibility Requirements

AI agents must meet specific requirements to integrate seamlessly with the infrastructure:

1. **Docker Images**: Agents must have Docker images that can either be built from their GitHub repositories or pulled from DockerHub.

2. **Local Dataset Processing**: Agents must process datasets as local files mounted into the container. This ensures secure data handling, as opposed to relying on external APIs or cloud-hosted datasets.

4.4.1 Autocoderover (ACR)

Autocoderover (ACR) [24] is a versatile AI agent designed to tackle real-world software engineering tasks. Its configuration (`agent_config`) is tailored to ensure seamless integration with our pipeline. The agent is hosted on GitHub, and its Docker image can be built using the provided repository or specified directly by the user. ACR relies on a structured workflow, starting with pre-run commands to build its Docker image and prepare the runtime environment. ACR supports multiple LLMs, and we ran it with **GPT-4o**. Table 4.2 gives an overview of the unique configuration keys used for ACR.

ACR's `docker_container_commands` include vital preparatory steps. These commands ensure that necessary files, such as `tasks.txt` and `swe_tasks.json`, are created inside the container. It activates the required Conda environments (`swe-bench` and `auto-code-rover`) and sets up essential configurations. These steps are critical for aligning ACR with the mounted SWE-bench (secret, in our context) dataset and pipeline dependencies.

The agent's main execution occurs via the `run_command`, which is customizable using placeholders interpolated by the pipeline at runtime. For example, the {`model_name`} and {`config_file_path`} parameters are dynamically substituted with their respective values. Outputs are generated in the specified directory, which is mapped from the host to the container to maintain compatibility across environments.

Additionally, we employed Docker's volume mappings to make ACR integrate seamlessly withour pipeline. These mappings allow files like `run_setup.py` and `constants.py` — both modified for our pipeline's requirements — to be mounted into the container. Notably, ACR downloads the SWE-bench repository when the container has been started. We mapped our constants file — containing the new installation steps - to the constants file in the container, ensuring that ACR references our file during execution.

We also made a minor change in `utils.py`: The GitHub URL "`https://{token}@github.com/swe-bench/`" was edited to "`https://{token}@github.com/swe-bench-secret`", which is our private organization. In contrast to ACR, Swe-agent pulls directly from the ac-

tual GitHub URLs. For example, SWE-agent can pull from `https://github.com/django-/django` rather than `https://{token}@github.com/swe-bench/django/django`.

Two modifications were made to `setup.py`: The first was to implement logic for accessing the dataset locally rather than from Huggingface. The second was the addition of installation steps, specifically addressing the new "modification_steps" logic. This feature will be made available to future AI engineers who wish to test their agents.

Table 4.2: Unique Items in Autocoderover (ACR) Configuration

Key	Actual Value	Comment
`docker_container_commands`	Commands to create necessary files, activate environments, and set up configurations.	Derived from the agent's documentation. Includes tasks setup, Conda activation, and running `run_setup.py`.
`run_command`	`PYTHONPATH=. python app/main.py swe-bench --model {model_name} --setup-map ../SWE-bench/setup_result/setup_map.json --tasks-map ../SWE-bench/setup_result/tasks_map.json --output-dir output --task-list-file /opt/SWE-bench/tasks.txt`	Primary command for running the agent. Placeholders like {`model_name`} are interpolated.
`predictions_output_message`	`SWE-bench input file created:`	Used to locate the output files.
`env_vars`	`AZURE_OPENAI_API_KEY`, `ENDPOINT_URL`, `AZURE_OPENAI_API_VERSION`	Environment variables interpolated at runtime.
`volume_mappings`	Paths mapped from host to container, such as `run_setup.py`, datasets, and constants files.	Ensures compatibility with pipeline-modified SWE-bench files.

4.4.2 Swe-Agent

Swe-Agent [21] is another AI agent developed by the creators of SWE-bench. Unlike ACR, it is designed with minimal container preparation, relying on pre-built Docker images. It supports multiple LLMs, and we ran it with **GPT-4o**.Table 4.3 outlines the unique configuration keys used for Swe-Agent.

Swe-Agent's workflow begins with `pre_run_commands`, which include pulling the latest version of the Docker image from DockerHub (`sweagent/swe-agent:latest`) and building a new image locally (`sweagent_image`). This streamlined approach ensures that the agent remains updated and ready for execution without requiring additional setup within the container.

The agent's primary execution occurs through the `run_command`, which dynamically interpolates placeholders like {`model_name`}, {`per_instance_cost_limit`}, and {`data_path`} at runtime. This enables seamless integration with various datasets and configurations.

Swe-Agent outputs its predictions to a specified directory, which is mapped between the host and container for compatibility. The prediction output is identified using the `predictions_output_message`, ensuring proper tracking of results.

To enhance compatibility with our pipeline, Docker's volume mappings are extensively utilized. Critical files and directories, such as `swe_secret_lite.json`, `keys.cfg`, and `default.yaml`, are mounted into the container. For instance, the dataset path `swe_secret_lite.json` is mapped to the container path `/app/datasets/swe_secret_lite.json`, and is referenced in the run command as data_path, enabling the agent to process local datasets directly. Similarly, the configuration file `default.yaml` and output directory (`trajectories`) are appropriately mapped for seamless execution.

Notably, we introduced PYTHONPATH as an environment variable with SWE-agent: `PYTHONPATH=/app`. This is because rather than locally cloning downloading SWE-bench into its container, SWE-agent uses it as a package. We generally found it easier to do this because we effectively tell the container to look to our version of SWE-bench first when searching for the SWE-bench module. This includes the entire harness folder which houses the constants file containing installation instructions. This ensures proper integration with our pipelines constants and other configuration files, and eliminates the need for hard-coded paths within the agent in this context.

SWE-agent's `swe_env.py` is the only file that had to be slightly modified. Two modifications were made to it: We included the "modification_steps" logic, and we pointed mirror installations to our GitHub organization, rather than SWE-bench's. Again, the modification_steps function and an example of how it is used will be provided in the appendix.

Table 4.3: Unique Items in Swe-Agent Configuration

Key	**Actual Value**	**Comment**
pre_run_commands	`docker pull sweagent/swe-agent:latest`, `docker build -t sweagent_image .`	Commands to pull and build the Docker image.
docker_container_commands	(empty)	No additional commands are required to prepare the container.
run_command	`python run.py --model_name {model_name} --per_instance_cost_limit {per_instance_cost_limit} --config_file {config_file_path} --data_path {data_path} --skip_existing`	Primary execution command for Swe-Agent. Placeholders are dynamically interpolated.
predictions_output_message	Saved predictions to	Message printed upon prediction generation.
volume_mappings	Paths to datasets (`swe_secret_lite.json`), configurations (`default.yaml`), and trajectories.	Enables data access and output generation.

4.5 Evaluation Results

The system produces detailed evaluation reports summarizing agent performance on the SWE-bench-secret dataset. These reports include anonymized logs and metrics reflecting the agent's ability to resolve software engineering tasks. Anonymization ensures that sensitive repository details remain confidential.

Chapter 5

Results

This chapter presents the outcomes of evaluating AI agents on the SWE-secret-lite dataset. The focus includes the dataset's construction, agent performance, and a comparative analysis with public datasets. Additionally, we analyze the unintended consequences of model-generated patches and investigate the reasons behind failures, providing insights into areas requiring improvement.

5.1 SWE-secret-lite Construction

We adopted SWE-bench's[12] approach to handling time and cost constraints by creating a lite version of our hidden benchmark. The SWE-secret-lite dataset was meticulously designed to ensure tasks remained evenly distributed across the 3 repositories. Even though our dataset is hidden to researchers, we are aware that models might have come across some of the GitHub issues we collect to form our tasks. Therefore, for our evaluation set, we prioritized tasks from 2024, a period beyond the cutoff date of most models' training data. Unlike SWE-bench-lite, which includes no tasks from 2024, SWE-secret-lite incorporates exactly 29 tasks from this year.

To balance the dataset, an equal number of tasks (29) from 2023 and 2022 were included, providing diversity while maintaining compatibility with SWE-bench's structure. We observed that 27 out of the 29 tasks from 2024 belonged to Repo1. Therefore, to avoid bias, tasks from 2023 and 2022 were carefully selected to exclude Repo1. As a result, while 27 of the 29 tasks selected for 2024 belonged to Repo1, Repo2 and Repo3 were prioritized for the earlier years, creating a diverse yet balanced distribution across recent years.

Ironically, while we had prioritized tasks from 2024 to challenge the agents with truly hidden GitHub issues, the only successfully resolved tasks also originated from this year. This finding raises several intriguing possibilities about how AI agents interact with and resolve tasks introduced after their training cutoff date. For example, one hypothesis could be that newer tasks, although technically beyond the models' training data horizon, still adhere to patterns or conventions that remain consistent with previous development practices, making them easier to address than anticipated. Conversely, it may indicate that earlier tasks, despite being within the models' nominal training scope, incorporate nuances, deprecated syntax, or less standardized coding practices that the agents struggle to navigate.

This outcome also brings to light the complexity of defining and identifying truly "unseen" tasks. It may not be enough to rely solely on temporal boundaries; factors such as repository domain, codebase complexity, codebase guidelines, and adherence to modern standards might influence an agent's ability to solve a given problem. As a result, these observations suggest that future research should investigate not only temporal aspects but also the qualitative characteristics of tasks — such as their level of documentation, code style consistency, and complexity of dependencies.

In essence, this paradoxical result opens up new avenues for understanding agent performance. It encourages researchers to explore more granular indicators of "unseenness" beyond the simple cutoff date, to refine evaluation methodologies, and to better understand how agents generalize.

5.2 Evaluation of AI Agents

The evaluation revealed nuanced differences in agent performance. SWE-Agent successfully resolved one instance in version 2.10 of Repo1. In contrast, ACR resolved two instances: one in version 2.10 of Repo1 and another in version 3.0 of Repo1. Notably, no tasks from Repo2 and Repo3 were resolved by either agent, emphasizing the challenge of addressing issues in diverse repositories.

When compared to public datasets, SWE-secret-lite results underscore the potential impact of training data exposure. SWE-Agent achieved a 1.72% resolution rate on SWE-secret-lite, significantly lower than its 18.33% performance on public datasets. Similarly, ACR's resolution rate dropped from 19.00% on public data to 3.45% on SWE-secret-lite. Despite this decline, ACR consistently outperformed SWE-Agent, mirroring trends observed in public benchmarks. The reliability of SWE-secret-lite as a standard for evaluating agents on genuinely private tasks is confirmed by these results.

Table 5.1: Task Distribution for ACR in SWE-secret-lite by Year

Year	Total Instances	Resolved Instances	Resolution Rate (%)
2024	29	2	6.90
2023	12	0	0.00
2022	17	0	0.00

Table 5.2: Task Distribution for SWE-Agent in SWE-secret-lite by Year

Year	Total Instances	Resolved Instances	Resolution Rate (%)
2024	29	1	3.45
2023	12	0	0.00
2022	17	0	0.00

5.3 Insights from Pass-to-Fail Regressions

We define regressions as situations where tests that previously passed ended up failing after the agents' prediction patches were applied, signifying a decline in the code quality of the overall codebase. Our pass-to-fail analysis evaluated the unintended consequences of model-generated patches by measuring regressions, where tests that previously passed failed after applying the patch.

- **Overall Rates and Severity:** ACR introduced pass-to-fail cases in 18.75% of its patches, while SWE-Agent had a comparable rate of 18.18%. However, SWE-Agent's regressions impacted more tests on average, with 10.17 regressions per affected instance compared to ACR's 6.22.
- **Total Regressions:** ACR caused 56 regressions across all instances, while SWE-Agent introduced 61, despite producing fewer patches overall. This indicates a higher density of regressions in SWE-Agent's patches.
- **Trade-offs Between Coverage and Stability:** ACR produced more patches, offering broader coverage but at the expense of introducing regressions in a greater number of instances. SWE-Agent generated fewer patches but with higher regression severity.

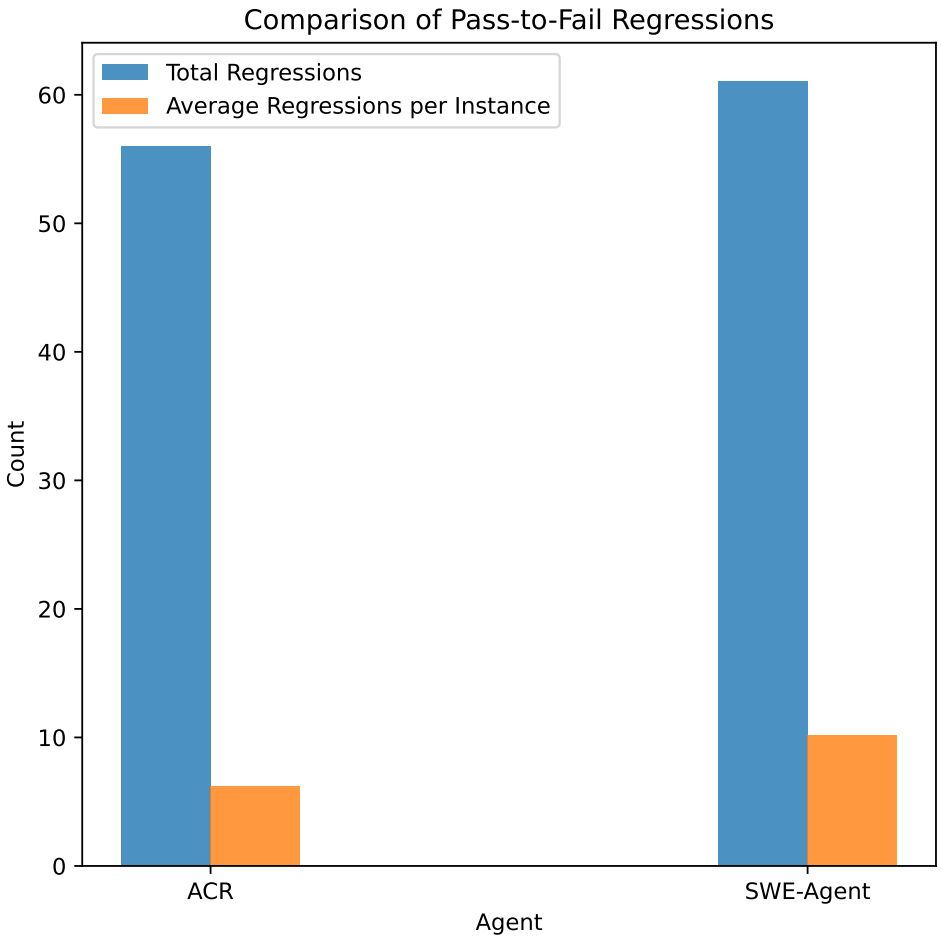

Figure 5.1: Comparison of Pass-to-Fail Regressions Across Agents

5.4 Analysis of Failure Reasons

To further analyze the agents' shortcomings, failure reasons were categorized based on predefined scenarios. Significant logic deviations from the gold patch were the dominant reason for failures, accounting for 66.7% in ACR and 73.0% in SWE-Agent.

ACR had a higher rate of incomplete handling of test cases (21.6%) compared to SWE-Agent (10.8%). SWE-Agent exhibited a broader range of failure types, including syntax

errors and edge case failures. Failures due to incorrect library usage were minimal in both agents, demonstrating moderate competence in selecting appropriate tools.

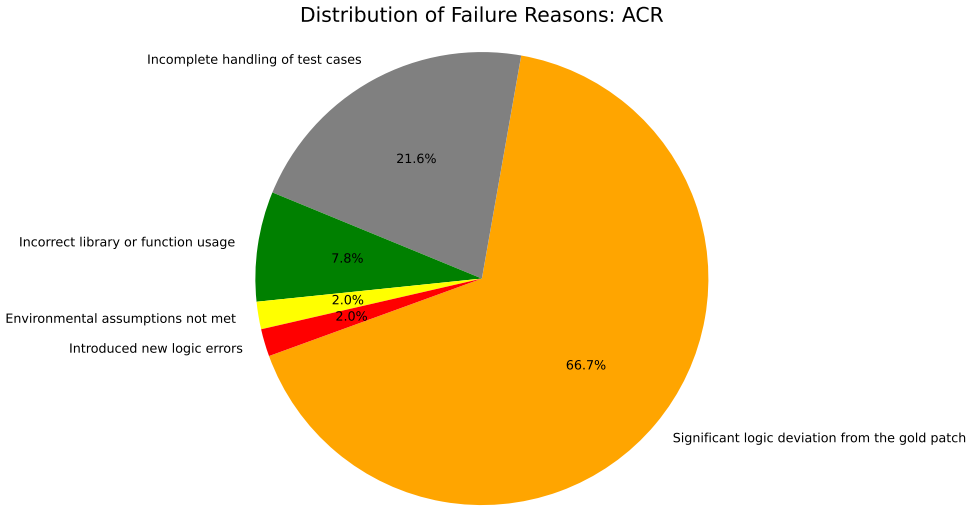

Figure 5.2: Distribution of Failure Reasons: ACR

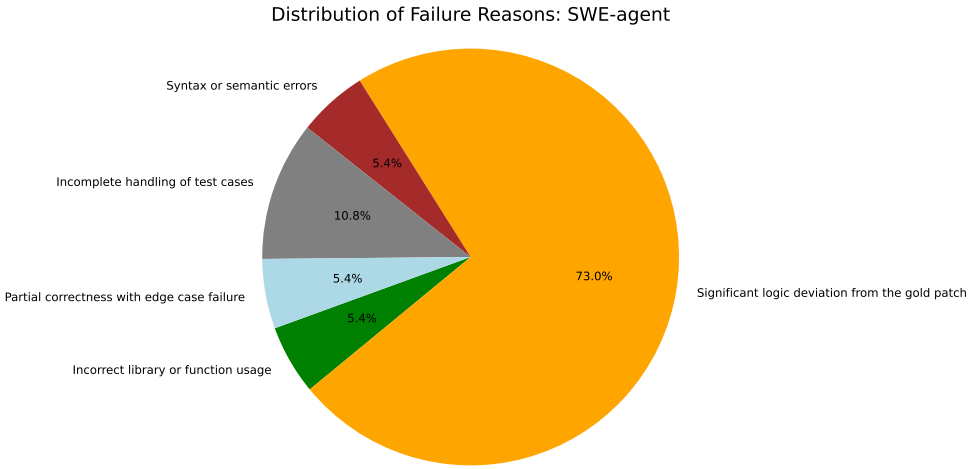

Figure 5.3: Distribution of Failure Reasons: SWE-Agent

5.5 Conclusions

The results highlight the limitations of current AI agents in resolving hidden tasks. Both ACR and SWE-Agent struggle with aligning their patches to gold standards, with logic deviations and incomplete test coverage being primary issues. ACR offers broader coverage but with greater regressions, while SWE-Agent shows higher regression severity despite generating fewer patches.

These findings emphasize the need for improved strategies to enhance logical coherence and test coverage, particularly in handling tasks that fall outside the familiar patterns derived from public training data. The SWE-secret-lite dataset proves to be a potent benchmark, offering valuable insights into the limitations and strengths of AI agents.

Chapter 6

Threats to Validity

This chapter describes the threats to the validity of our research results, categorized into construct, internal, and external validity.

6.1 Construct Validity

A primary threat to construct validity is the reliance on test-based metrics to evaluate agent performance. While these metrics provide quantitative insights, they fail to capture qualitative aspects like code readability, maintainability, or adherence to best practices. This limitation may overlook patches that are logically correct but fail due to minor discrepancies in test setups.

Another potential threat is dataset contamination, where agents may have been inadvertently trained on publicly available testing data from SWE-bench or SWE-bench-lite, thereby artificially boosting their performance. To mitigate this, the SWE-Secret-Lite dataset was constructed using repositories that remain inaccessible to researchers, this ensures that all tasks were genuinely new to the agents. By prioritizing tasks from 2024, we aimed to challenge the agents with data beyond their training scope. However, the agents' ability to resolve only these 2024 tasks suggests that further exploration of diverse time periods and task types is necessary to fully assess their generalizability.

6.2 Internal Validity

Internal validity concerns potential biases in the experimental setup that could influence results. Errors in task curation or repository modifications pose significant risks. To mitigate this, we validated all task instances in the SWE-Secret-Lite dataset, ensuring proper installation, execution, and alignment with their problem statements.

6.3 External Validity

External validity relates to the extent to which findings generalize beyond the study's scope. The SWE-Secret-Lite dataset includes tasks from three repositories selected for their popularity and testing practices, which may not represent the full range of software engineering challenges. While this selection offers valuable insights, results might not generalize to projects with differing architectures or coding styles.

The evaluations focus on ACR and SWE-Agent, agents compatible with the SWE-bench framework. Their performance may not extrapolate to other AI systems, particularly those trained on distinct datasets or employing different methodologies. Additionally, while GPT-4-based evaluations are reflective of state-of-the-art systems, the findings might vary with alternative models or frameworks.

Chapter 7

Future Works and Conclusion

7.1 Conclusion

In this study, SWE-bench-secret, a new and secret benchmark for evaluating AI agents on private software engineering tasks taken from popular GitHub repositories, is presented. SWE-bench-secret places a higher priority on data secrecy than publicly accessible benchmarks in order to reduce the dangers of contamination and overfitting.

The structure and diversity of SWE-bench are mirrored in the development of SWE-secret-lite, which provides a lightweight but demanding substitute for preliminary assessments. With much lower resolution rates than public datasets, evaluations on SWE-secret-lite demonstrate the shortcomings of existing AI agents. The tasks' subtle complexity was difficult for both SWE-Agent and Autocoderover to handle, highlighting the need for enhancements in logical coherence, in AI agents.

We introduce a framework that allows researchers to submit their agents for evaluation. The infrastructure guarantees a smooth and secure integration for researchers and practitioners by automating the submission process and anonymizing sensitive data.

SWE-bench-secret establishes itself as a strong benchmark for advancing the capabilities of AI agents in software engineering, laying a foundation for future enhancements, including the development of verified datasets and more comprehensive evaluations.

7.2 Future work

The SWE-bench-secret dataset represents a significant step forward in evaluating AI agents on private, uncontaminated, real-world software engineering tasks. However, several opportunities for future work remain.

First, following the footsteps of SWE-bench and SWE-bench-Java, we aim to develop a verified [15] version of the dataset. This version will involve careful curation and validation of tasks by experienced Python developers, ensuring an even higher standard of relevance and quality.

Currently, only 58 out of 457 task instances in SWE-secret-lite have been evaluated due to time and cost constraints. We plan to expand this work by evaluating the remaining tasks and subsequently testing on the verified subset. This approach will provide a comprehensive understanding of the dataset's potential to benchmark AI agents effectively.

In addition, we plan to continually collect new tasks from the same repositories while ensuring that the data is entirely novel and outside the training data of current models. This involves focusing on post-training data periods (e.g., tasks from 2024 onward). By doing so, we aim to enhance the dataset's ability to evaluate AI agents on challenges that truly reflect their capacity to generalize to unseen, real-world scenarios.

Finally, we hope to add more repositories to the dataset while preserving its secrecy. The integrity of the benchmark and the strength of its evaluation system will be maintained while a wider range of software engineering challenges are covered.

References

[1] Usman Anwar, Abulhair Saparov, Javier Rando, Daniel Paleka, Miles Turpin, Peter Hase, Ekdeep Singh Lubana, Erik Jenner, Stephen Casper, Oliver Sourbut, et al. Foundational Challenges in Assuring Alignment and Safety of Large Language Models. *arXiv preprint arXiv:2404.09932*, 2024.

[2] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program Synthesis with Large Language Models. *arXiv preprint arXiv:2108.07732*, 2021.

[3] Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. Multipl-E: A Scalable and Extensible Approach to Benchmarking Neural Code Generation. *arXiv preprint arXiv:2208.08227*, 2022.

[4] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374*, 2021.

[5] Yangruibo Ding, Zijian Wang, Wasi Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, et al. CrossCodeEval: A Diverse and Multilingual Benchmark for Cross-File Code Completion. *Advances in Neural Information Processing Systems (NeurIPS)*, 36, 2024.

[6] GitHub. GitHub Copilot: Your AI Pair Programmer. https://github.com/features/copilot. Accessed: 2024-12-04, Publisher: GitHub.

[7] Shahriar Golchin and Mihai Surdeanu. Time Travel in LLMs: Tracing Data Contamination in Large Language Models. *arXiv preprint arXiv:2308.08493*, 2023.

[8] Alex Gu, Baptiste Rozière, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I Wang. CruxEval: A Benchmark for Code Reasoning, Understanding, and Execution. *arXiv preprint arXiv:2401.03065*, 2024.

[9] IBM. What are AI Agents? https://www.ibm.com/think/topics/ai-agents/. Accessed: 2025-01-16, Publisher: IBM.

[10] IBM Research. What is Retrieval-Augmented Generation? https://research.ibm.com/blog/retrieval-augmented-generation-RAG/. Accessed: 2025-01-16, Publisher: IBM Research.

[11] Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and Contamination-Free Evaluation of Large Language Models for Code. *arXiv preprint arXiv:2403.07974*, 2024.

[12] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-Bench: Can Language Models Resolve Real-World GitHub Issues? *arXiv preprint arXiv:2310.06770*, 2023.

[13] Tianyang Liu, Canwen Xu, and Julian McAuley. RepoBench: Benchmarking Repository-Level Code Auto-Completion Systems. *arXiv preprint arXiv:2306.03091*, 2023.

[14] OpenAI. Introducing ChatGPT. https://openai.com/index/chatgpt. Accessed: 2024-12-04, Publisher: OpenAI.

[15] OpenAI. Introducing SWE-Bench Verified. https://openai.com/index/introducing-swe-bench-verified/. Accessed: 2024-12-10, Publisher: OpenAI.

[16] Yonatan Oren, Nicole Meister, Niladri Chatterji, Faisal Ladhak, and Tatsunori B Hashimoto. Proving Test Set Contamination in Black-Box Language Models. *arXiv preprint arXiv:2310.17623*, 2023.

[17] Manley Roberts, Himanshu Thakur, Christine Herlihy, Colin White, and Samuel Dooley. To the Cutoff... and Beyond? A Longitudinal Perspective on LLM Data Contamination. In *The Twelfth International Conference on Learning Representations (ICLR)*, 2023.

[18] Oscar Sainz, Jon Ander Campos, Iker García-Ferrero, Julen Etxaniz, and Eneko Agirre. Did ChatGPT Cheat on Your Test?, 2023.

[19] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741*, 2024.

[20] Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-Based Software Engineering Agents. *arXiv preprint arXiv:2407.01489*, 2024.

[21] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. *arXiv preprint arXiv:2405.15793*, 2024.

[22] Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. CERT: Continual Pre-Training on Sketches for Library-Oriented Code Generation. *arXiv preprint arXiv:2206.06888*, 2022.

[23] Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, et al. SWE-bench-java: A GitHub Issue resolving Benchmark for Java. *arXiv preprint arXiv:2408.14354*, 2024.

[24] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous Program Improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 1592–1604, 2024.

[25] Kun Zhou, Yutao Zhu, Zhipeng Chen, Wentong Chen, Wayne Xin Zhao, Xu Chen, Yankai Lin, Ji-Rong Wen, and Jiawei Han. Don't Make Your LLM an Evaluation Benchmark Cheater. *arXiv preprint arXiv:2311.01964*, 2023.