Optimizing AI Agents for Automated Software Engineering with Palimpzest

by

Jason Li

S.B. Computer Science and Engineering, Massachusetts Institute of Technology, 2024

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2025

Authored by:	Jason Li Department of Electrical Engineering and Computer Science May 1, 2025
Certified by:	Michael Cafarella Principal Research Scientist at CSAIL, Thesis Supervisor
Accepted by:	Katrina LaCurts Chair Master of Engineering Thesis Committee

Optimizing AI Agents for Automated Software Engineering with Palimpzest

by

Jason Li

Submitted to the Department of Electrical Engineering and Computer Science on May 1, 2025 in partial fulfillment of the requirements for the degree of

MASTER OF ENGINEERING IN ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

ABSTRACT

The deployment of large language models (LLMs) as autonomous agents is transforming the software development landscape. Increasingly more engineers are using natural language agents to expedite and guide development workflows, while large organizations are investing heavily on building agentic systems for tasks such as code generation and code repair. A key challenge in developing such systems is tuning agent hyperparameters— settings that affect performance such as choice of model, temperature settings, and context window sizes. As system complexity grows, the hyperparameter space expands, complicating optimization under real-world compute and time constraints. In this work, we present Palimpzest[1] as an agentic optimizer able to balance cost and performance objectives by tuning agentic hyperparameters. We demonstrate that Palimpzest can tune our agent hyperparameters at 8.5 times lower cost and with 24 times greater time efficiency compared to the conventional grid search. By integrating our custom-built Debugger and Code Editor Agents as new operators within Palimpzest, we enhance the system's ability to resolve real-world GitHub issues. And to facilitate hyperparameter selection, we also introduce File Coverage, Report Accuracy, and Patch Similarity along with the traditional SWE-Bench Score as quality evaluation methods used by Palimpzest's optimization loop. When evaluated on the SWE-Bench Lite[2] benchmark, our optimized system achieves a 15% score at a significantly lower cost compared to previous approaches.

Thesis supervisor: Michael Cafarella

Title: Principal Research Scientist at CSAIL

Acknowledgments

I would like to express my sincere gratitude to my thesis advisor, Michael Cafarella, for his continuous support, thoughtful guidance, and for welcoming me into the vibrant research community of CSAIL's Data Systems Group. The opportunity to collaborate with such an inspiring group of researchers has been one of the most enriching aspects of my academic journey. I am especially thankful for the trust he placed in me and the invaluable learning experiences he made possible throughout this past year.

I am also deeply grateful to Matthew Russo for his extraordinary mentorship and steadfast support in the development of this project. As one of the principal architects behind Palimpzest, Matthew has been instrumental not only in shaping the technical foundation of my work, but also in guiding me through countless obstacles with patience, clarity, and generosity. His dedication, insight, and genuine care have left a lasting impact on me.

I would also like to extend my heartfelt thanks to Professor Tim Kraska for co-advising me during the second semester of my MEng. His support played a key role in helping me grow as both a researcher and an engineer, and I am truly grateful.

To the many members of the Data Systems Group I've had the privilege of working with, including Ziniu Wu, Ferdinand Kossman, and Amadou Latyr Ngom, thank you for your insights, encouragement, and kindness. Your collaborative spirit has made this journey all the more fulfilling.

I am also incredibly thankful for the friends who have supported me throughout my time at MIT. A special thanks to the MIT Lion Dance Club for being a source of joy, camaraderie, and some of my most cherished memories.

Most of all, I would like to thank my family. To my two sisters who have always had my back, whether they admit it or not, thank you for your love and presence in my life. And to my mom, my greatest inspiration and fiercest supporter, your endless sacrifices, efforts, and love have carried me further than I could have ever imagined. None of this would be possible without you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In recent years, the adoption of language model (LM) coding assistants has grown rapidly. From conversational agents like ChatGPT and Claude to development tools such as Cursor and Copilot, AI-driven systems have become embedded in modern software engineering workflows. In parallel, many organizations have begun developing sophisticated multi-agent frameworks designed to address software engineering tasks in an end-to-end fashion. To evaluate the effectiveness of such systems, benchmarks like SWE-Bench have been introduced, focusing on real-world bug-fixing scenarios. We will use SWE-Bench to evaluate our own system. Tasks in SWE-Bench are often nontrivial, frequently requiring a comprehensive understanding of the code base and involving coordinated edits across multiple functions, classes, and files. Despite these challenges, systems such as SWE-Agent[3], OpenHands[4], and the Globant Code Fixer Agent[5] have demonstrated the efficacy of LM-based agentic architectures for addressing complex software engineering problems. These systems typically employ a modular design, where specialized agents are responsible for subtasks such as bug localization or precise code editing. This agentic paradigm effectively emulates the behavior of human developers by navigating the code base, diagnosing issues, and iteratively applying fixes, making it a suitable approach for automated software engineering tasks.

1.1.1 Challenges

As agentic systems grow in complexity, engineers are increasingly challenged with selecting appropriate agentic hyperparameters. We define agentic hyperparameters as configurable parameters that govern the behavior and performance of individual agents within a system. These may include choices such as the underlying language model for each agent, the number of reasoning iterations permitted, and constraints on token usage. Given the substantial cost of AI inference, these hyperparameters have a direct impact on the overall runtime cost. For example, GPT-4o incurs a charge of 5 USD per one million input tokens. In large code bases, where individual files can span thousands of tokens, a single multi-agent debugging session can result in several dollars of compute expenditure. This cost is exacerbated by long reasoning trajectories and the absence of strict token limits. Complicating matters further, new models are being released at a rapid pace, each with distinct capabilities, context lengths, and pricing structures. Engineers must also tailor agent configurations based on context: development settings may favor faster, low-cost configurations for iteration and testing, whereas production scenarios may justify more expensive setups for higher accuracy. Manually tuning these parameters for each use case is both time-consuming and expensive. Consequently, there is a pressing need for automated mechanisms to efficiently optimize agentic hyperparameters in response to varying workloads and constraints.

Our Approach. In this work, we extend the functionality of Palimpzest[6] to support the automated optimization of agentic hyperparameters. Palimpzest is a semantic programming framework that enables users to write high-level AI workflows for diverse applications, including tasks such as filtering real estate listings and detecting fraudulent emails. It achieves this through a set of composable semantic operators that process and reason over input data. Crucially, Palimpzest incorporates a built-in optimization layer that selects system configurations based on user-specified objectives like cost or quality. This allows the system to dynamically adapt hyperparameters to suit varying workload requirements. The core insight driving Palimpzest is that "machines, not human engineers, should decide how best to optimize semantic analytics applications." In the context of automated code repair,

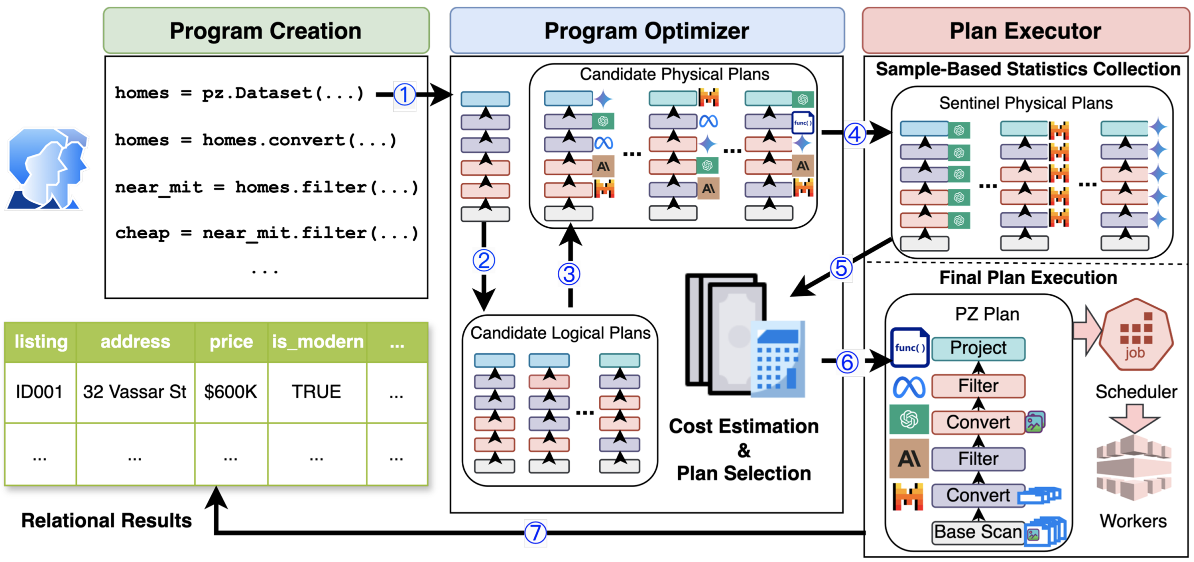


Figure 1.1: The Palimpzest Architecture[1]

we leverage this insight to eliminate the manual effort and trial-and-error often required to configure agent systems, instead allowing Palimpzest to automatically discover effective hyperparameter settings under budget or performance constraints.

1.1.2 SWE-Bench

We will be evaluating our system using the SWE-Bench[2] benchmark. SWE-Bench is a challenging evaluation suite designed to assess the capabilities of automated systems in performing real-world software engineering tasks. The dataset consists of historical GitHub issues, each containing a description of the issue and the repository it originated from. These issues are drawn from a curated selection of large, actively maintained Python repositories, spanning diverse domains and coding styles. The goal of a system evaluated on SWE-Bench is to autonomously generate a patch that fixes the bug described in the issue, using only the information available at the time the issue was filed. This requires the system to understand the context of the code base, reason about the root cause of the problem, and apply precise edits that lead to a correct and test-passing solution.

1.1.3 Palimpzest

To understand our contributions, it is essential to have a basic understanding of how Palimpzest operates. Palimpzest's[6] recent paper details it's optimization algorithm in detail, but for the purposes of this paper we will provide a high level description. Figure 1.1 illustrates the optimization process Palimpzest follows to select the most suitable execution plan for a given workload. Users begin by writing high-level programs for their workload via the Palimpzest Python API. In the example shown, the user defines a dataset of home listings and applies a sequence of convert and filter operations to refine the dataset to match specific criteria. Palimpzest parses this user-defined workflow into a set of logical plans, where each plan is composed of a sequence of logical operators, each representing a semantic operation such as filtering or transformation. These logical operators are akin to steps in a recipe, representing the task to be executed.

Each logical plan is then expanded into multiple physical plans, which correspond to concrete implementations of the logical operators, similar to the physical tools required to execute a step in a recipe. The number of physical plans depends on the number of ways each operator can be implemented, each accounting for different hyperparameter settings. The Plan Executor samples and executes these candidate physical plans using inputs drawn from a validation dataset. During this sampling process, the cost and quality of each physical operator is evaluated to develop cost and quality profiles used during the final plan selection. Quality can be defined by the user, while cost is measured in terms of dollar compute costs. The optimization loop proceeds until a user-defined sampling budget is met. Palimpzest then selects the plan that best satisfies the user's optimization objective, such as maximizing quality, minimizing cost, or achieving the best quality under a fixed cost constraint. This final selected plan is then applied to the full input dataset to generate the desired output records. The key benefit of Palimpzest's optimization approach is in its ability to approximate the quality and cost of each physical plan while accommodating user defined budgets and avoiding costly exhaustive search procedures.

1.2 Contributions

In this paper we:

- Demonstrate Palimpzest's ability to identify optimal agentic hyperparameters for a set of language model agents, achieving performance comparable to configurations obtained through expensive grid search procedures.
- Extend Palimpzest's capabilities to resolve real-world GitHub issues by introducing two custom agents as logical operators. The first is a Debugger Agent that specializes in diagnosing the root cause of the issue, and the second is a Code Editor Agent that modifies code and generates a final output patch based on the Debugger Agent's analysis. These agents cooperate to carry out multi-step repair workflows. We evaluate the enhanced Palimpzest system on the SWE-Bench benchmark, demonstrating performance gains compared to the previous non-agentic Palimpzest implementation.
- Present experimental results on SWE-Bench Lite demonstrating that our optimized agentic system performs complex software engineering tasks at a level comparable to high performing solutions, but at a significantly lower cost.
- Introduce File Coverage, Report Accuracy, and Patch Similarity along with the traditional SWE-Bench score as quality metrics to improve Palimpzest's optimization of our agentic system. We analyze how each metric influences hyperparameter selection and impacts system performance.

Chapter 2

Related Works

The recent application of large language models (LLMs) to software engineering tasks has led to an increase in research demonstrating the effectiveness of AI agents. High-performing LLMs such as OpenAI's GPT-4 series and Anthropic's Claude have shown strong capabilities in code generation. These powerful models have been employed to address a broad range of complex software engineering challenges, including repository-level feature implementation and intricate bug resolution. Since the introduction of benchmarks like SWE-Bench and HumanEval[7], agentic systems have achieved impressive results on complex code generation tasks. These systems extend LLM capabilities by assigning them specialized roles, providing structured instructions, and integrating tool access, thereby enabling multi-step planning and iterative reasoning in code generation workflows.

To demonstrate that agentic systems can be effectively tuned using Palimpzest, our multi-agent framework draws inspiration from state-of-the-art (SOTA) agentic code repair systems. In this section, we (1) review several modern agentic systems and the key design features that influenced the architecture of Palimpzest's agents, and (2) compare how these systems approach hyperparameter selection relative to Palimpzest's optimization-based methodology.

2.0.1 Modern Agentic Systems

An early example of an agentic system is SWE-Agent[3], which demonstrated strong performance on SWE-Bench by equipping a single language model (LM) agent with a custom agent-computer interface (ACI). SWE-Agent's primary focus was on designing an effective ACI to fully leverage the reasoning capabilities of LLMs for resolving GitHub issues. Rather than relying on traditional interfaces such as the Linux shell, SWE-Agent introduces LM-friendly abstractions and streamlined commands for file search, navigation, and editing- paired with concise and structured feedback. This ACI also enhances agent reliability by incorporating syntax-aware editing, guardrails to prevent invalid actions, and context management to reduce the memory footprint of irrelevant information. At each reasoning step, SWE-Agent "generates a thought and a command, then incorporates the feedback from the command's execution." This iterative reasoning loop is based on the ReAct[8] prompting framework, an extension of the Chain-of-Thought paradigm[9], which allows LMs to interact with external tools and respond to environment feedback in a structured and deliberate manner.

In contrast to SWE-Agent's single-agent design, the Globant Code Fixer Agent[5] adopts a multi-agent strategy built on similar principles of integrating custom tools and functionalities. Globant's system achieves a SWE-Bench score of 48.33% using a structured two-stage pipeline. The first stage, localization, employs specialized agents equipped with advanced tools to navigate the code base, identify faulty files, and produce a detailed report pinpointing the root cause of the bug. This stage leverages techniques such as dependency analysis, fuzzy search, and language identification to improve the accuracy of fault localization. In the second stage, fixing, three dedicated agents collaborate: the Architect Agent proposes candidate solutions based on the localization report; the Editor Agent implements the changes with high precision; and the Critic Agent reviews the modifications, applying a retry mechanism to iteratively refine the patch when necessary. Each of these agents has access to a diverse set of tools, including `search_codebase`, `read_file`, `get_related_files`, and `edit_file`, enabling robust and targeted behavior across the pipeline.

CodeAgent[10] targets repository-level code generation by focusing on designing agentic tools that emulate the workflow of human software engineers. Agents are equipped with tools such as `WebSearch`, `DocSearch`, and `CodeInterpreter`, enabling them to search the web for relevant information, retrieve pertinent documentation, and execute code within the repository's runtime environment. These capabilities allow agents to acquire domain-specific knowledge, modify existing code, and verify changes, closely mirroring the behavior of human developers working in large codebases. Ablation studies for CodeAgent highlight the importance of a code symbol navigation tool, which grants agents access to the source code of specific classes and functions. Similar tools are incorporated into our Palimpzest agent architecture. At the time of its release, CodeAgent outperformed commercial systems such as GitHub Copilot on its own benchmark.

In contrast to the previously discussed systems, HyperAgent[11] represents one of the first generalist coding agents capable of addressing a broad range of software engineering tasks, from GitHub issue resolution on SWE-Bench to repository-level code generation on RepoExec[12]. Like CodeAgent, HyperAgent aims to emulate the workflow of human developers, but it adopts a multi-agent architecture composed of four specialized agents: the Planner, Navigator, Code Editor, and Executor. The Planner Agent interprets human prompts, coordinates the activities of other agents, and formulates a resolution strategy. The Navigator Agent handles information retrieval within the codebase, using tools such as `code_search` and `go_to_definition`. The Editor Agent performs code modification and generation across multiple files, leveraging tools like `code_search` and `open_file`. Finally, the Executor Agent validates solutions and reproduces issues through interaction with a simulated bash shell environment. This structured decomposition allows HyperAgent to scale across diverse tasks by delegating responsibilities to specialized agents.

Other systems, such as CodeR[13], have specifically targeted the challenges inherent in multi-agent architectures. These challenges include information loss between agents, non-progressive reasoning loops, and difficulties in agent coordination under complex task plans. CodeR addresses these issues by introducing a task graph data structure designed to streamline agent behavior. This structure constrains the number of possible actions at each step, front-loads planning to simplify downstream decision making, and mitigates memory limitations commonly encountered by LLMs. The system comprises five specialized agents: a Manager for plan generation, a Reproducer for generating test cases to reproduce issues, a Fault Localizer for identifying relevant code, an Editor for applying code modifications, and a Verifier for executing tests to validate solutions. These agents utilize a suite of tools, including `search_method`, `search_class`, `find_file`, and `edit_file`, among others, to support their respective tasks.

To showcase Palimpzest's effectiveness in optimizing agent hyperparameters, our agentic architecture adopts agents and tools commonly found in the systems discussed above. While we do not exhaustively implement every technique used across these systems, we aim to show that Palimpzest's hyperparameter tuning allows us to achieve comparable results with standard techniques at a significantly lower cost. Similar to HyperAgent, Globant's Code Fixer Agent, and CodeR, Palimpzest's architecture includes a Debugger Agent responsible for fault localization and a Code Editor Agent tasked with performing code modifications. We also incorporate widely used tools present in nearly all of these systems, including `summarize_file`, `extract_method`, `extract_class`, `search_codebase`, `get_file_content`, and `edit_file`. In addition, our agents leverage the ReAct prompting strategy, which has been successfully employed in Globant, CodeR, CodeAgent, and SWE-Agent, enabling iterative reasoning and tool invocation guided by intermediate thoughts and feedback.

2.0.2 Hyperparameter Selection

Each of the aforementioned systems required careful selection of agent hyperparameters. At minimum, all systems had to determine the most suitable language models for their agents. For instance, HyperAgent, which consists of four specialized agents, conducted extensive and computationally expensive experiments to identify the best LLM for each of its agents. SWE-Agent, in addition to model selection, evaluated a broader range of parameters, including agent context window size, history processing configurations, and decoding temperature. They conducted a grid search over four hyperparameters, each with two candidate values, resulting in 16 unique configurations. Using a subset of 37 SWE-Bench instances and averaging 592 evaluations, with an estimated resolution cost of \$1.92 per instance at an 18% success rate, the total cost of this grid search can be reasonably estimated at approximately \$600. Furthermore, SWE-Agent also benchmarked various tool implementations, including editors, search utilities, and file viewers, on SWE-Bench Lite, which likely incurred significant computational expenses. These examples illustrate that most existing systems rely on costly grid search procedures to identify optimal agent configurations.

Palimpzest's ability to automatically optimize hyperparameters directly addresses the challenge of selecting hyperparameters. Instead of having to perform a grid search, our optimization algorithm intelligently enumerates, samples, and evaluates various agentic implementations across the hyperparameter space. This allows systems to evaluate a larger space of hyperparameters for the same cost. In our results, we demonstrate that Palimpzest is able significantly reduce the time and cost of hyperparameter search compared to the traditional grid search. Moreover, Palimpzest is able to select hyperparameters that optimize for the cost of execution while demonstrating high performance on SWE-Bench.

Chapter 3

Methods

3.1 Agent Architecture

3.1.1 Overview

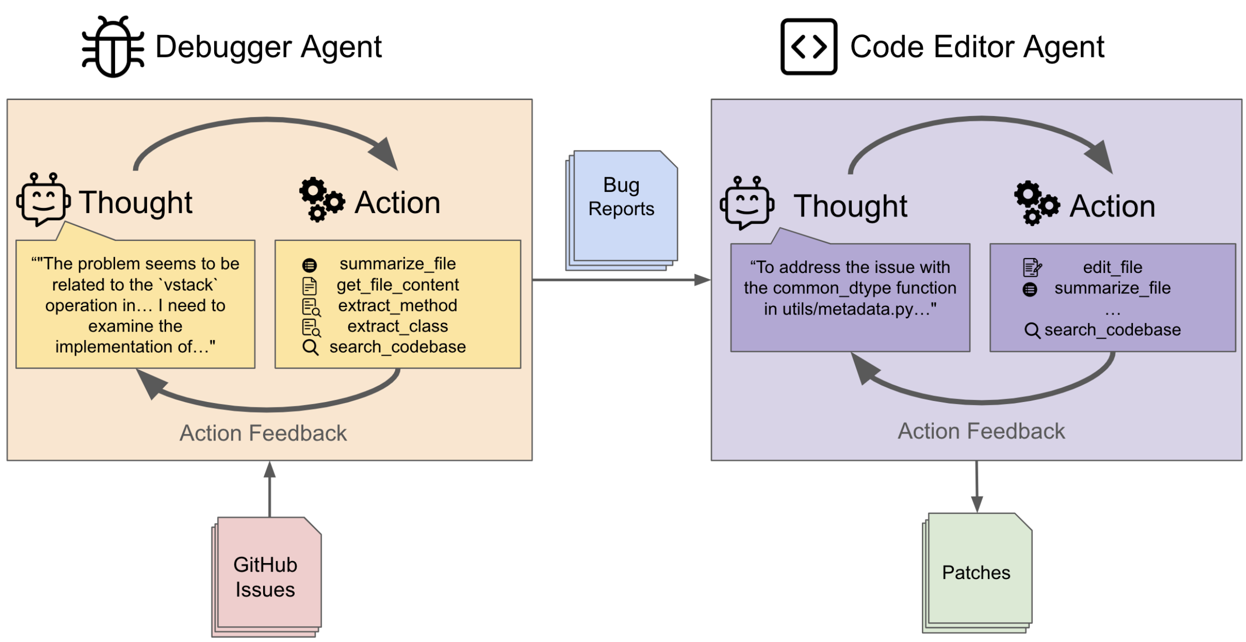


Figure 3.1: High-level Diagram of Agent Architecture

Prior to integrating agents into the Palimpzest optimization framework, we first construct the agents that comprise our system. As illustrated in Figure 3.1, the system includes two custom built agents: a Debugger Agent and a Code Editor Agent. The Debugger Agent is tasked with identifying the root cause of a given GitHub issue and producing a detailed report that specifies both the location and rationale for the problem. The Code Editor Agent then utilizes this report to implement the necessary code modifications across the repository, ultimately returning a patch that addresses the issue.

Both agents are configured to employ the ReAct prompting technique, reasoning through sequences of thoughts and actions. Each agent has access to a set of specialized tools tailored to its specific task. As previously mentioned, these tools were selected based on common tools across various agentic systems. At each step, the agent generates a thought that informs its subsequent action. An action is defined as an invocation of a tool. After a sufficient number of iterations, the agent produces its final output. This separation between debugging and code editing enables each agent to concentrate its reasoning trajectory on a more focused objective, allowing for higher quality outputs.

3.1.2 Tool Construction

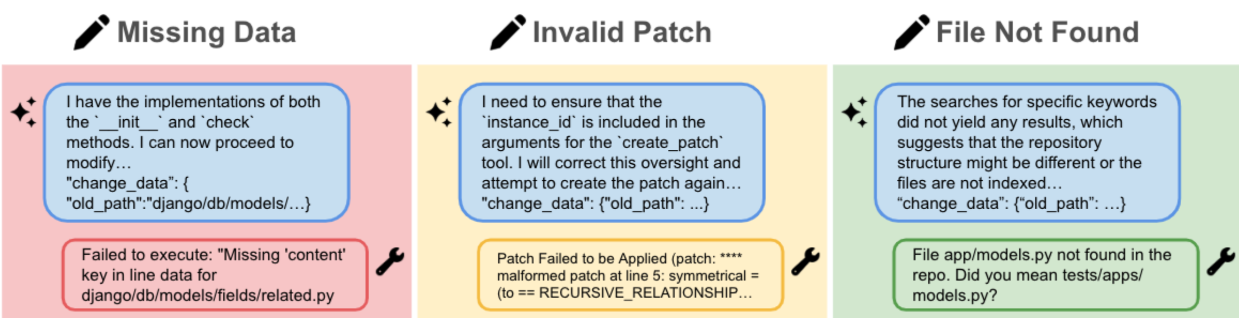


Figure 3.2: Edit File Tool Feedback Examples

In constructing tools and their descriptions, we must keep in mind that language models face several inherent limitations that hinder their effectiveness on complex software engineering tasks. These include hallucinations, where the model generates plausible but incorrect information; limited context windows, which constrain the amount of code or documentation the model can consider at once; and the haystack problem, where relevant information is deeply buried within its prompt history, making it difficult for the model to retrieve and reason over the correct context. These challenges are particularly pronounced in large, multi-file code bases, where understanding dependencies, control flow, and subtle bug patterns often requires integrating information spread across many of lines of code.

Description:
Searches the entire codebase for the provided keyword, returning the files the keyword appears in.
Provide `repo_name` in `"owner/repo"` format.

Guidelines:

- If searching for a function or class definition, it may be useful to prefix the keyword with `"def "` or `"class"`.
- Do not search for vague keywords that may excessive results, such as `"test"`, `"function"`, or `"def"`. Search for more specific phrases instead like `"def function_name(self, arg1, arg2)"`
- Only the first 15 results will be returned, so be specific.

Usage Tips:

- Use this tool to determine the location of a function, class definition, or member variable.

Figure 3.3: The `search_codebase` Tool Description Provided to Agents

Given these limitations, we follow several guidelines in constructing tools and their descriptions:

- **Limit the number of tools per agent.** Providing an agent with too many tools can hinder its ability to reason effectively about which tool to invoke, often leading to the selection of suboptimal tools.
- **A single generalized tool is preferable to multiple specialized tools.** In line with the previous guideline, it is often more efficient to design one comprehensive tool rather than several narrowly focused subtools. This approach reduces the total number of tools and simplifies agent decision making. For example, a tool that returns both class and function names from a file is more effective than maintaining separate tools

for each.

- **Tools should provide rich and informative feedback.** An effective toolset minimizes the number of iterations required for an agent to complete its task. To achieve this, tools should return detailed feedback capable of correcting flawed reasoning or inputs while reinforcing correct ones. Such feedback helps guide the agent's subsequent thoughts in a productive direction and prevents it from deviating toward unproductive paths. Figure 3.2 provides examples of feedback that the `edit_file` tool provides for varying edits.

- **Tool feedback should be concise and limited in length.** Concise feedback is easier for the agent to interpret and contributes to more efficient use of the prompt context, thereby reducing hallucinations and improving information retention. When tools access large amounts of code, it is especially important to restrict the number of tokens returned. For instance, as illustrated in Figure 3.2, only the malformed line needs to be specified in the case of an invalid patch, rather than the entire patch.

- **Keep tool descriptions concise.** Concise tool description and naming helps the agent to confidently identify which tool to use. Figure 3.3 illustrates the `search_codebase` tool description as an example.

- **Include usage guidelines in tool descriptions.** Tool descriptions should provide clear usage guidelines, including instructions on proper input formatting to prevent parsing errors, as well as recommendations on when a tool should or should not be used. Figure 3.3 provides an example of usage tips and guidelines provided to the agent for the `search_codebase` tool.

Debugger Agent Example Reasoning Trajectory

Problem Statement
"Python code printer not respecting tuple with one element... I ran into an issue with the Python code printer..."

↓

Thought 1
"The issue seems to be related to how the Python code printer in SymPy 1.10 handles tuples with a single element. To investigate this, I need to locate the relevant code... that deals with the code generation for tuples..."

↓

Action 1
search_codebase("def lambdify")
["sympy/utilities/lambdify.py", "sympy/utilities/tests/test_lambdify.py"]

↓

Thought 2
"The ‘lambdify‘ function is located in the ‘sympy/utilities/lambdify.py‘ file... I need to examine the classes and methods within this file..."

↓

...

↓

Thought 8
"The ‘lambdastr‘ function is responsible for converting expressions to a string that can be evaluated to a lambda function...I should investigate the ‘_recursive_to_string‘ function to understand how it handles tuples with a single element."

↓

Action 8
extract_method("_recursive_to_string")
def _recursive_to_string(doprint, arg):...

↓

Bug Report
"The root cause of the bug is in the ‘_recursive_to_string‘ function... The issue occurs because the function does not add a comma for single-element tuples.. To fix this issue, the ‘_recursive_to_string‘ function should be modified ... Here is a suggested fix:..."

Figure 3.4: Debugger Agent Reasoning Trajectory for SWE-Bench sympy__sympy-23262

3.1.3 Debugger Agent

Given a GitHub issue containing a description of the problem, the objective of the Debugger Agent is to produce a detailed bug report that identifies the root cause of the issue and explains how it can be resolved, referencing specific functions, classes, and file names. The report also describes how the bug was discovered and which files, functions, or classes were examined during the investigation. To support the generation of this report, the Debugger Agent is equipped with a set of tools designed to facilitate navigation and comprehension of large code repositories. These tools were selected based on their prevalence and demonstrated importance in prior agentic systems.

The tools are as follows:

- **summarize_file(file_name)**: Returns a structured JSON object containing all classes in the specified file, along with their member functions and any standalone functions. If the exact file path is not provided, the tool performs a fuzzy match across the repository and selects the closest match within a predefined similarity threshold. If no match meets the threshold, the agent is informed that the summary has failed and is provided with the most similar file name as a suggestion.

- **get_file_content(file_name, starting_line_number, include_line_numbers)**: Returns the content of the specified file beginning at the given line number. The agent may optionally request that line numbers be included, depending on whether they are needed for tasks such as patch generation or simply for viewing the code. This tool is limited to returning a maximum of 200 lines per invocation. Accessing additional content requires a subsequent call with a different starting line number. This constraint helps reduce the inclusion of irrelevant code in the agent's context. Prior to displaying the content, the agent is also informed of the total number of lines in the file.

- **extract_method(file_name, class_name, function_name, include_line_numbers)**: Returns the code for a function in the given file and class. The class_name argument helps to prevent the incorrect function from being returned in the case that there are multiple functions with the same name (e.g. __init__). Similar to other tools that access files, fuzzy matching is also used to find the file.

- **extract_class(file_name, class_name, include_line_numbers)**: Returns the code for a class in the given the file.
- **search_codebase(keyword)**: Returns all files for which the given keyword appears. The agent is instructed to search for precise phrases and to avoid vague words such as "def". This prevents an excessive number of files from being returned.

Code Editor Agent Example Reasoning Trajectory

Bug Report
"The root cause of the bug is in the '_recursive_to_string' function..."

↓

Thought 1
"I need to locate the '_recursive_to_string' function in the 'sympy/utilities/lambdify.py' file to apply the suggested fix. First, I will search for the function definition to confirm its location in the file."

↓

Action 1
search_codebase("def _recursive_to_string")
["sympy/utilities/lambdify.py"]

↓

...

↓

Action 2
extract_method("_recursive_to_string")
"948": "def _recursive_to_string(doprint, arg):", "949": " """Functions in lambdify accept both SymPy types and non-SymPy types such as python", "950": " lists and tuples. This method ensures that we only call the doprint method of the... ",

↓

Thought 3
"I have extracted the current implementation of the '_recursive_to_string' function. Now, I will create a patch to modify this function by adding a comma for single-element tuples, as suggested in the bug report."

↓

Action 3
edit_file("sympy/utilities/lambdify.py")
diff –git a/sympy/utilities/lambdify.py b/sympy/utilities/lambdify.py...

↓

Patch

```
diff --git a/sympy/utilities/lambdify.py b/sympy/utilities/lambdify.py
--- a/sympy/utilities/lambdify.py' +++ b/sympy/utilities/lambdify.py ...
```

Figure 3.5: Code Editor Agent Reasoning Trajectory for SWE-Bench sympy__sympy-23262

3.1.4 Code Editor Agent

Given a bug report from the Debugger Agent, the task of the Code Editor Agent is to generate an output patch that resolves the original GitHub issue. This patch may consist of multiple hunks, sections of modified code, distributed across several files. Patch generation demands precision and is particularly susceptible to errors. For each hunk, the agent must specify the original and new start line numbers, the lengths of the corresponding code segments, a context header, and a line-by-line designation of additions, deletions, or context lines. Agents frequently make mistakes in line number calculations, indentation, or syntax. In many cases, a patch will fail to apply if its indentation does not match the original code. These challenges are further amplified when the patch spans multiple hunks or files. Therefore, it is essential to design tools that address these specific issues and enable the agent to iteratively refine and improve the patch.

In order to minimize the number of tools, the Code Editor is equipped with a tool to make changes in addition to the default navigation tools:

- **edit_file(change_content)**: This feature rich tool enables the agent to iteratively modify a file and construct a valid output patch by addressing key challenges in patch generation. The agent provides a change_content object in patch format that specifies the intended code modifications. The tool first verifies the line counts for each hunk and corrects them if necessary, then minimizes the patch to reduce its size and mitigate the risk of application errors. Minor issues are automatically resolved to prevent redundant tool invocations. If ambiguity or missing information is detected such as a line lacking a context, addition, or deletion tag, the tool prompts the agent to revise the patch. After minimization, the patch is applied in a sandboxed environment to validate its formatting, including indentation and context alignment. If an error occurs, the tool identifies the problematic hunk and line number. If successful, the tool performs a syntax check on the modified files and reports any compilation errors. Otherwise, it returns the finalized patch to the agent.

3.1.5 Palimpzest Agent Integration

```python
import Palimpzest as pz

def buildSWEBenchPlan(data_path):
    # Create Data Reader to structure GitHub Issues
    datareader = GithubIssueDataReader(data_path)

    # Create Dataset object to store GitHub Issues
    github_issues = pz.Dataset(datareader)

    # Process GitHub Issues with the Debugger and Code Editor Agents
    bug_reports = github_issues.add_agent(agent_name="debugger")
    code_patches = bug_reports.add_agent(agent_name="code_editor")

    # Return the generated code patches
    return code_patches
```

Figure 3.6: Python Implementation of Palimpzest's Declarative Agentic Workflow

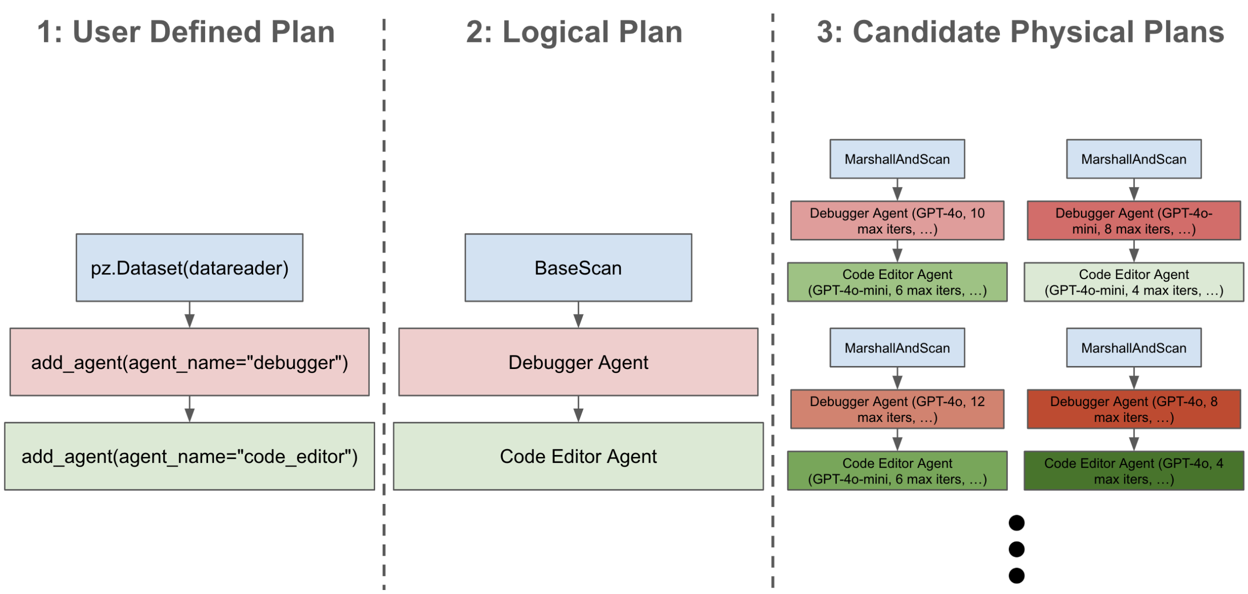


Figure 3.7: Illustration of How Palimpzest's Algorithm Translates a (1) user defined agent plan into a (2) logical plan and finally, to a (3) set of candidate physical plans.

In order to enable the optimization of our agentic workflow, we first integrate the agents into Palimpzest as logical operators. As illustrated in Figure 3.6, we introduce a new `add_agent` API that enables users to incorporate agents into their Palimpzest plans. We then implement the Debugger and Code Editor agents as logical operators within Palimpzest, serving as abstract representations that will later be transformed into concrete physical implementations. Next, we implement the physical operators for the Debugger and Code Editor Agents containing the implementation details of our agents. This involves selecting identifiers for the operators and exposing the hyperparameters of our implementation to the optimizer. Finally, we define the transformation rules required to translate the Debugger and Code Editor logical operators into their numerous corresponding physical expressions.

Figure 3.7 illustrates how Palimpzest's algorithm uses the `add_agent` API to generate a logical plan from the user defined plan. This logical plan is then translated a set of candidate physical plans by the aforementioned transformation rules. Palimpzest's optimizer then uses its sampling algorithm to measure the quality of the numerous physical plans against inputs from a validation dataset. A final optimal physical plan is then chosen based on quality and cost objectives.

We followed several key guidelines in determining which agent hyperparameters to expose to the Palimpzest optimizer for tuning:

- **The parameter should have a meaningful influence on both cost and quality.** This ensures that the user can explore a diverse space of cost and quality tradeoffs when specifying plan objectives, such as maximizing quality under a fixed cost constraint.
- **The optimal value of the parameter may depend on the values of other hyperparameters.** This interdependence makes such parameters more challenging to tune. By leveraging Palimpzest as an optimizer, we can identify the optimal configuration across these interacting hyperparameters.
- **The parameter should be easily discretizable.** Most agent hyperparameters are already discrete by nature, which aligns well with Palimpzest's design, as it performs optimization over a discrete search space.

For the purpose of demonstration, we also selected common hyperparameters that many state of the art systems spent a significant amount of time and money tuning manually.

Each agent has the following hyperparameters:

- **Max Iterations**: The max iterations hyperparameter defines the maximum number of reasoning steps permitted for each agent. If set too high, agents may overreason, unnecessarily increasing the cost of addressing a bug. If set too low, agents may be unable to complete their tasks. Appropriately tuning this parameter also acts as a safeguard against repetitive reasoning loops. This is a critical hyperparameter, as its optimal value depends on the tools available to the agent, the nature of its task, and interactions with other hyperparameters. Moreover, it has a significant influence on overall execution cost.

- **Model**: The model hyperparameter specifies which language model the agent uses. Model selection has a substantial impact on both cost and quality, as different models exhibit varying performance characteristics. Additionally, each model has its own token limit, further influencing the agent's ability to reason over large contexts. As such, this is a crucial hyperparameter to tune.

- **Context Size**: The context_size hyperparameter defines the maximum number of past iterations retained in an agent's memory. If set too large, the agent may struggle with effective recall, which is essential for tasks such as debugging and patch generation. Additionally, larger context sizes significantly increase the number of tokens processed per request, raising model usage costs. Conversely, a context size that is too small may omit critical historical information, impairing the agent's ability to reason effectively.

3.1.6 Agentic Quality Metrics

As previously discussed, Palimpzest evaluates the quality of sampled physical operators to guide the selection of agent hyperparameters. If no quality metric is explicitly defined by the user, Palimpzest defaults to an equivalence-based metric: an output is assigned perfect quality if it is identical to the provided validation label, and zero otherwise. However, in the context of patch generation and the SWE-Bench benchmark, this strict equivalence metric is impractical. Although the benchmark provides a reference patch, many distinct patches may correctly resolve the same bug. Consequently, we implement and integrate a range of alternative quality metrics better suited for agent optimization in this setting.

Although it is possible to tune the hyperparameters for both the Debugger and Code Editor by combining them into a single agentic operator, this approach substantially increases the size of the hyperparameter search space and prevents tailored optimization for each agent. To address this, we define separate quality metrics specific to the unique outputs of each agent, enabling more targeted and efficient tuning.

To measure the quality of the Debugger Agent's bug report, we use two measures of quality:

- **File Coverage**: File coverage measures the fraction of files identified in the bug report as requiring modification that are actually altered by the ground truth patch. This metric rewards hyperparameter configurations that effectively locate the files relevant to the issue.

- **Report Accuracy**: Report accuracy is a language model supported quality metric that prompts a language model to evaluate how well the suggested changes in a bug report align with the ground truth patch. This enables the optimizer to reason semantically about the content of the report. A high accuracy score indicates that the bug report is consistent with and justifies the changes made in the patch, while a low score suggests that the report fails to describe or support the modifications reflected in the patch.

To measure the quality of the Code Editor Agent's patch, we use two measures of quality:

- **SWE-Bench Score**: This metric provides a binary signal, returning 1 if the patch successfully resolves the issue and 0 otherwise. While it serves as the most direct indicator of success when aiming to maximize SWE-Bench performance, its utility is limited by the sparsity of positive signals under constrained sampling budgets. Moreover, the computational overhead associated with executing the SWE-Bench evaluation harness significantly increases the runtime of optimization. As a result, we also explore alternative quality metrics that can approximate patch quality without incurring these limitations.

- **Patch Similarity**: This metric leverages a language model to compare the agent generated patch with the validation patch and assign a similarity score. The language model is instructed to evaluate similarity based on functional behavior: a score of 1 indicates that the patches are functionally equivalent, while a score of 0 reflects entirely divergent intentions. Intermediate values, such as 0.5, correspond to patches that share similar goals but differ in implementation or behavior.

Chapter 4

Experimental Results

4.1 Experimental Setup

Research Questions

In our experiments, we aim to investigate the following research questions:

1. How effective is Palimpzest at tuning agent hyperparameters under budget constraints compared to the commonly used exhaustive grid search?
2. How do the quality metrics used to evaluate the Debugger and Code Editor Agents influence hyperparameter selection and, consequently, the overall performance of the system?
3. When implemented with standard techniques, can our system achieve performance comparable to existing approaches at a significantly lower cost?
4. Does the agentic Palimpzest system outperform the naive Palimpzest baseline?
5. What is the impact of excluding certain hyperparameters from the optimization process?
6. What is the behavior of our agent under optimal hyperparameter settings?

These questions are designed to evaluate the cost-effectiveness and performance potential of using Palimpzest to tune an agentic system built with standard SWE-Bench techniques. Accordingly, we assess success based on low optimization and evaluation cost and performance comparable to existing systems.

Dataset Preparation. Due to compute constraints, we primarily evaluate our system on SWE-Bench Lite a reduced version of the full benchmark consisting of 300 task instances drawn from 11 of the original 12 repositories. For experiments requiring extensive exploration of hyperparameter and quality metric combinations, we further restrict evaluation to a smaller subset of SWE-Bench Lite which we refer to as SWE-Bench Tiny. Such cases are noted explicitly in the experimental descriptions. SWE-Bench Lite is designed to preserve the diversity of the full benchmark. To integrate the dataset into Palimpzest, each instance is stored as a separate JSON file and loaded through a custom DataReader for processing by our agents.

Hyperparameter Options. For each hyperparameter, we define a set of candidate values from which the optimizer can select. These options are chosen based on a combination of qualitative observations of agent behavior, available compute resources, and constraints imposed by API access.

Debugger Agent Hyperparameters:

- **Model**: GPT-4o, GPT-4o-mini
- **Max Iterations**: 8, 10, 12 max iterations
- **Context Size**: Past 5 Iterations, All iterations

Code Editor Agent Hyperparameters:

- **Model**: GPT-4o, GPT-4o-mini
- **Max Iterations**: 4, 6, 8 max iterations
- **Context Size**: Past 3 Iterations, All iterations

Optimization Objective. For our experiments, we primarily use Palimpzest's MaxQuality-AtFixedCost objective which maximizes the quality of our plan at a user specified fixed cost. We set the fixed cost to 0.17 USD per instance for a total of 50 USD.

4.2 Palimpzest Can Find Optimal Hyperparameters

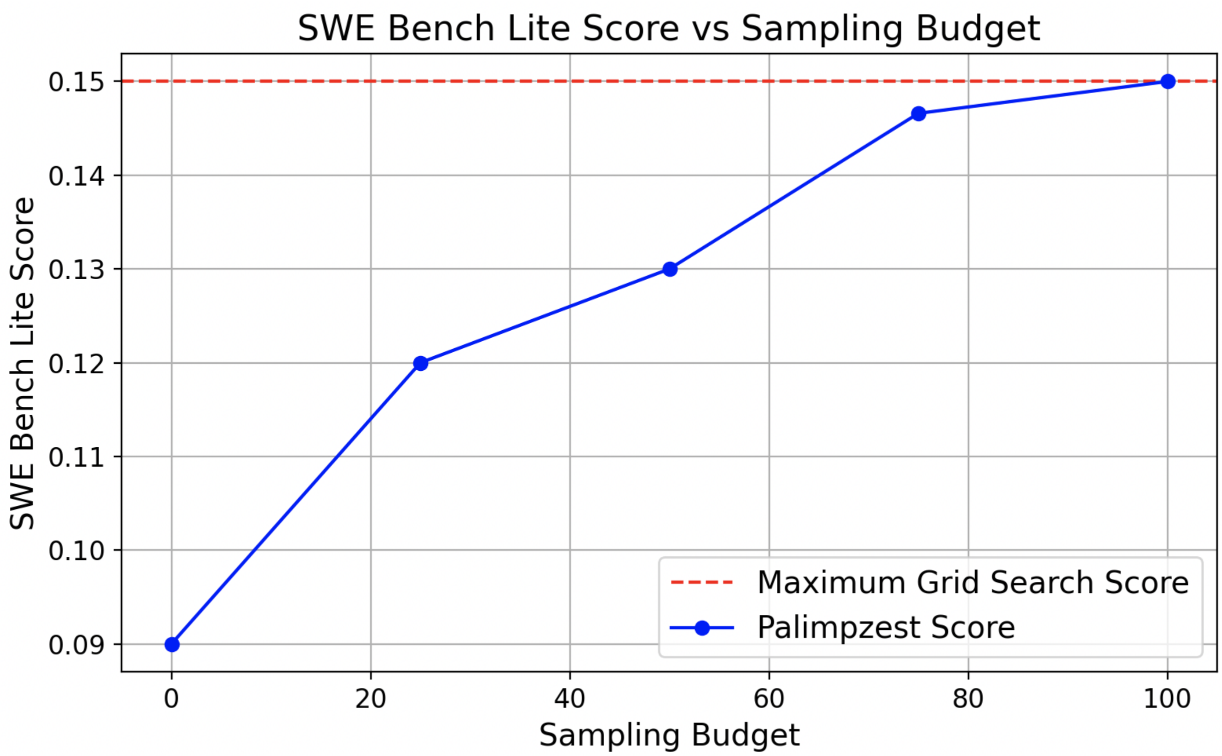


Figure 4.1: SWE-Bench Lite Performance as a Function of Palimpzest Sampling Budget

As shown in Figure 4.1, Palimpzest is capable of identifying hyperparameters that achieve performance comparable to the best configurations found through grid search. Due to computational constraints, the grid search was conducted on a subset of 50 instances from the SWE-Bench Lite dataset, after which the top-performing hyperparameters were re-evaluated on the full dataset. For the Debugger Agent, the optimal configuration was found to be GPT-4o with a maximum of 10 iterations and a context size of the five most recent iterations. For the Code Editor Agent, the best results were achieved using GPT-4o with 8 maximum iterations and a context size that included all previous iterations. Within a sampling budget of 100- meaning the optimizer was restricted to evaluating 100 distinct physical operator configurations on a single validation instance- Palimpzest successfully converged to the same optimal hyperparameters identified by grid search. It is important to note that larger sampling budgets may be necessary for agentic workflows involving more parameters. In our case, the total number of possible configurations was relatively small: 12 for the Debugger Agent and 12 for the Code Editor Agent, resulting in 24 total combinations.

Table 4.1: Comparison of Optimization Cost and Time Investment between Palimpzest and Grid Search

Method	Cost	Time Investment
Palimpzest	**$8.08**	**30 mins**
Grid Search	$68.88	12 hrs

Table 4.1 compares the optimization cost and time required to search for agent hyperparameters using Palimpzest versus grid search. Even with efforts to reduce grid search overhead by excluding unlikely hyperparameter combinations and limiting evaluation to a subset of the SWE-Bench Lite dataset, grid search remained approximately 8.5 times more expensive and 24 times more time intensive than optimization using Palimpzest.

4.3 Quality Metrics Improve Runtime and Performance

Table 4.2: SWE-Bench Tiny Performance Across Different Combinations of Quality Metrics Used for Tuning the Debugger and Code Editor Agents

		Code Editor Metric	
		SWE-Bench Score	Patch Similarity
Debugger Metric	File Coverage	0.13	0.12
	Report Accuracy	0.17	**0.18**

Table 4.2 presents a comparison of SWE-Bench performance across different combinations of quality metrics used during the optimization of the Debugger and Code Editor agents. Each evaluation is conducted on a subset of 100 instances from the 300 available in SWE-Bench Lite- which we refer to as SWE-Bench Tiny- using a sampling budget of 100. Among the evaluated configurations, the highest performing combination was achieved by applying Report Accuracy to the Debugger Agent and Patch Similarity to the Code Editor Agent.

Through qualitative observation, we find that File Coverage often under performs as a Debugger Agent quality metric because most hyperparameter configurations for the Debugger Agent are already capable of identifying the relevant files, resulting in homogeneous and uninformative quality scores. In contrast, Report Accuracy more effectively distinguishes between hyperparameters by evaluating not only whether relevant files are identified, but also whether the agent accurately reasons about the root cause of the issue and how it should be resolved.

We also observe that SWE-Bench Score and Patch Similarity tend to favor similar Code Editor hyperparameter configurations. This is expected, as a patch that passes the SWE-Bench test is also likely to receive a high similarity score. However, SWE-Bench Score produces sparser quality signals due to its binary nature, while Patch Similarity provides more granular feedback by awarding partial credit to patches that closely resemble the ground truth.

In practice, Patch Similarity offers a more efficient alternative to SWE-Bench Score for optimization. While the SWE-Bench harness incurs substantial runtime overhead due to requiring Docker image construction and cleanup for each evaluation, Patch Similarity is light-weight- significantly reducing evaluation time. Although it introduces a small API cost by issuing a single prompt per quality evaluation, it provides a faster and more flexible signal for guiding optimization.

4.4 Agentic Palimpzest Outperforms Naive Baseline

```
import Palimpzest as pz

def create_naive_plan(data_path):
    # Create Data Reader
    datareader = GithubIssueDataReader(data_path)

    # Create Dataset
    github_issues = pz.Dataset(datareader)

    # Localization of bug
    bug_locations = github_issues.sem_add_columns([
        {"name": "bug_location", "type": str, "desc": bug_location_desc},
        {"name": "relevant_files", "type": str, "desc": relevant_files_desc},
    ])

    # Generate a bug report
    bug_reports = bug_locations.sem_add_columns([
        {"name": "bug_report", "type": str, "desc": bug_report_desc},
    ])

    # Create patch
    patches = bug_reports.sem_add_columns([
        {"name": "model_patch", "type": str, "desc": patch_desc},
    ])

    patches.run(max_quality=True)
```

Figure 4.2: Python Implementation of Palimpzest's Declarative Agentic Workflow

Figure 4.2 shows the implementation of a naive Palimpzest SWE-Bench approach that does not utilize our agentic system. Using this naive implementation, we showcase the performance improvement gained from integrating our agents into Palimpzest. For this experiment, we use the oracle variant of SWE-Bench Lite as the system has no reliable means of navigating the code base and determining relevant files itself. The SWE-Bench Lite Oracle contains the relevant code files that were modified in the gold patch.

The evaluation begins by storing each instance in a dataset object. The system then performs bug localization, producing two new columns: `bug_location` and `relevant_files`. A bug report is subsequently generated to describe how the issue might be resolved, followed by the generation of a final patch. While we experiment with different column descriptions and prompt formulations, this naive approach fails to solve any instances from SWE-Bench Lite, yielding a score of 0%, even when provided with oracle data. The patches generated by the plan are often incoherent, frequently containing incorrect line numbers, inaccurate line counts, and improper formatting. The system's inability to adaptively reason, auto-correct, and leverage specialized tools highlights the critical advantage of our agentic framework. In comparison, the Debugger and Code Editor agents are equipped with rich tools and reasoning capabilities that enable substantially more effective performance.

4.5 Tuning Agent Hyperparameters Leads to Performance Gains

Table 4.3: This table compares the ablation of agent hyperparameters. Each value is the score received on a size 100 subset of SWE-Bench Lite for each hyperparameter excluded from Palimpzest's optimization framework. A fully optimized system achieves **0.18**.

Agent/Hyperparameter	Max Iterations	Model	Context Size
Debugger Agent	0.15	0.18	0.15
Code Editor Agent	0.11	0.12	0.16

Table 4.3 presents the results of ablation experiments in which individual hyperparameters of the Debugger and Code Editor Agents are set to default values rather than being optimized. Evaluated on 100 instances from SWE-Bench Lite, the fully optimized system achieves a score of 0.18, outperforming all partially optimized configurations except the one in which the model hyperparameter of the Debugger Agent is ablated. However, this default happens to coincide with the optimal setting. Notably, tuning the `max_iterations` hyperparameter of the Code Editor Agent proves especially critical as the default value of 4 iterations results in nearly a 40% reduction in performance compared to the fully optimized configuration. Overall, the fully optimized system consistently outperforms or matches all variants in which

any single hyperparameter is left unoptimized.

4.6 Palimpzest Achieves Competitive Performance at Reduced Cost

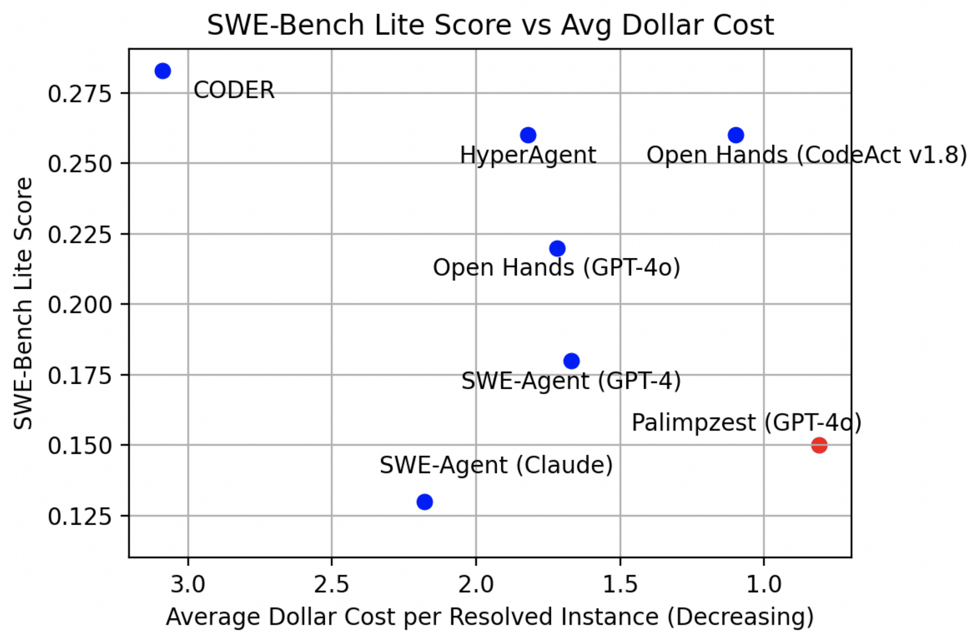


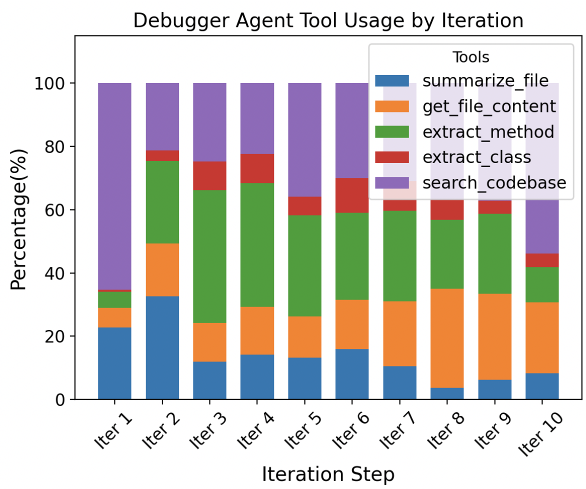
Figure 4.3: Average Cost and Performance Comparison Against Other SWE-Bench Systems

Figure 4.3 presents the average cost per resolved instance versus SWE-Bench Lite performance for a range of systems that report per-instance cost. The plot is based on explicitly published average dollar cost data for each system. Palimpzest achieves a SWE-Bench Lite score of 15% at an average cost of $0.81 per resolved instance. This outperforms SWE-Agent configured with Claude-3-5, both in terms of performance and cost-efficiency.

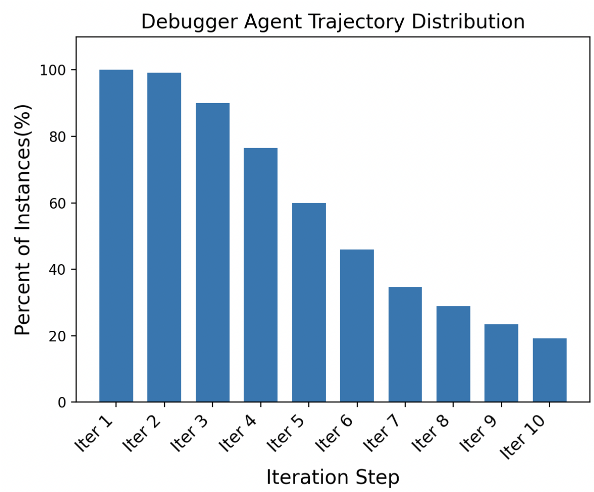
While other systems achieve higher benchmark scores, Palimpzest is significantly more cost-efficient than other high-performing agentic approaches. These results highlight Palimpzest's ability to select agent hyperparameters that satisfy strict cost budgets while preserving competitive performance.

4.7 Agent Behavior Under Optimal Hyperparameter Settings

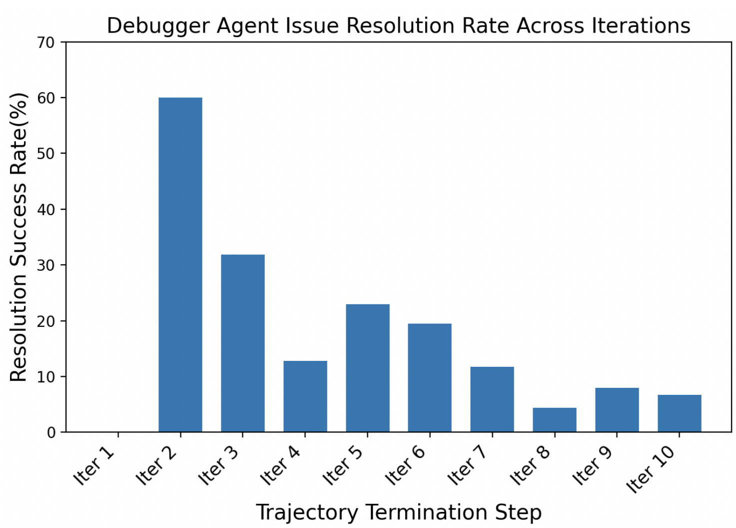
4.7.1 Debugger Agent



(a) Distribution of Tool Usage Across Reasoning Iterations



(b) Percent of Instances Across Reasoning Iterations



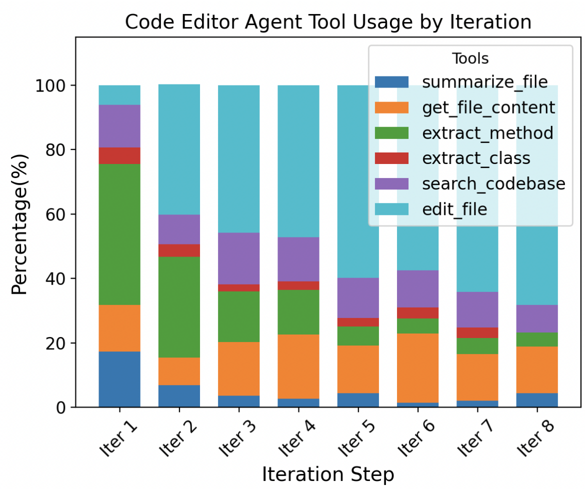
(c) This figure illustrates the resolution rate of GitHub issues for Debugger Agent trajectories that terminate at each specified iteration.

Figure 4.4: Debugger Agent Tool Usage and Issue Resolution Statistics.

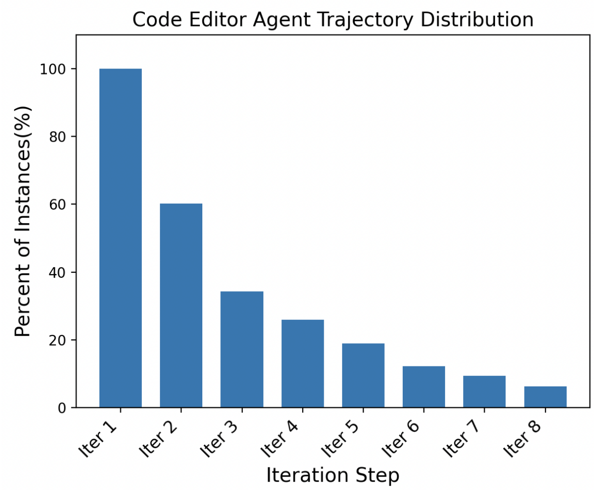
Figure 4.4 presents a high-level overview of the Debugger Agent's behavior across its reasoning iterations when set to optimal hyperparameters. The agent typically begins by searching the code base for keywords mentioned in the provided problem statement. When the relevant file is explicitly specified in the issue description, the agent may instead initiate the process by summarizing that file. During the early iterations, the agent primarily focuses on locating and interpreting the functions and files related to the issue, resulting in frequent use of the `extract_method` and `search_codebase` tools. Usage of the `get_file_content` tool increases in later iterations, often as a fallback strategy when more targeted searches fail to retrieve the necessary context, prompting the agent to inspect the entire file. The usage statistics of each tool demonstrates their individual utility, each contributing to the overall performance of the Debugger Agent.

From Figures 4.4b and 4.4c, we observe that fewer than 20% of instances reach the maximum of 10 reasoning iterations. Of those, less than 20% are ultimately resolved by the system. Figure 4.4c shows that longer trajectories are increasingly unlikely to be resolved, with the highest resolution rate occurring when the Debugger Agent terminates after just 2 iterations. In longer reasoning trajectories, the Debugger Agent typically fails to precisely localize the bug. In contrast, shorter reasoning traces often indicate higher agent confidence and correlate with greater resolution success. Immediate localization is rare, with the average number of reasoning steps per instance hovering around four. However, a substantial number of instances extend beyond the fourth iteration, with diminishing resolution rates.

4.7.2 Code Editor Agent



(a) Distribution of Tool Usage Across Reasoning Iterations



(b) Percent of Instances Across Reasoning Iterations

(c) This figure illustrates the resolution rate of GitHub issues for Code Editor Agent trajectories that terminate at each specified iteration.

Figure 4.5: Code Editor Agent Tool Usage and Issue Resolution Statistics.

Figure 4.5 presents a high-level overview of the Code Editor Agent's behavior across reasoning iterations. The agent typically begins by retrieving the relevant code segments described in the bug report generated by the Debugger Agent and identifying the corresponding line numbers that need to be modified to produce an accurate patch. This initial step frequently involves the use of the `extract_method` and `get_file_content` tools. Starting from the second iteration, usage of the `edit_file` tool increases significantly as the agent transitions to constructing the final output patch. This shows us that each tool used by the Code Editor is key in varying circumstances, contribution to the overall performance of the agent.

We observe that the Code Editor Agent generally requires fewer reasoning iterations than the Debugger Agent. In fact, 40% of instances successfully generate a final patch after just the first iteration. The number of instances terminating in later iterations decreases rapidly, with only 6% reaching iteration 8. As seen in Figure 4.5c, instances that require many iterations typically result in failing patches. This is often due to the agent generating patches that fail to adhere to the guidelines enforced by the `edit_file` tool, such as proper line numbers, indentation, or syntax. The most successful patches are generated in the second iteration, with a staggering 100% resolution rate. These patches are typically simpler to generate due to high quality bug reports provided by the Debugger Agent. Trajectories that terminate beyond the second iteration experience significantly lower success rates.

Many instances fail because the agents become trapped in reasoning loops, repeatedly invoking the same tools without meaningful progress. Recovery from such failures is only possible if the tools are capable of redirecting the agent away from incorrect behavior. This underscores the importance of designing tools that provide specific and informative feedback. In addition to improving overall agent reliability, such feedback reduces the number of reasoning iterations required to resolve an instance, thereby lowering the associated computational and monetary cost.

Chapter 5

Conclusion

In this work, we explore the application of Palimpzest as an agentic optimizer for our Debugger and Code Editor Agents. We show that Palimpzest is able to find the optimal set of agentic hyperparameters at 8.5x lower cost and with 24x the time efficient as compared to the conventional grid search. We also improve on the quality evaluation of the Palimpzest optimizer by introducing File Coverage, Report Accuracy, and Patch Similarity as quality metrics- improving overall system performance. Finally, we evaluate the performance of our system against SWE-Bench Lite, achieving a score of 15% at a significantly lower price point in comparison to other state of the art systems.

5.1 Challenges

The challenges faced during this work highlight important considerations for optimizing and building agentic systems:

- **Selecting hyperparameters.** Under a limited sampling budget, increasing the number of hyperparameters negatively impacts the quality of optimization by expanding the search space. Given the many possible ways to implement our agents, it was not always clear which hyperparameters would yield the most benefit when exposed to the optimizer.

- **Patch generation is difficult for language models**. It is unlikely that a language model will generate a properly formatted patch on the first attempt. Even minor formatting issues such as an incorrect line count or a misplaced newline character can cause the patch to fail during application. As a result, the design of patch generation tools required careful analysis and engineering. Built-in guardrails were necessary to detect and correct common errors made by the language model, ensuring that the resulting patches could be successfully applied to the code base.

- **Building Language Model based quality metrics**. During the implementation of language model based quality metrics- specifically Report Accuracy and Patch Similarity- we encountered challenges in ensuring the consistency of quality scores. Language models frequently returned varying scores for the same bug report or patch across different evaluations. To mitigate this variability, we developed detailed prompts that included examples and explicit scoring guidelines, which significantly improved response consistency and reliability.

5.2 Recommended Improvements

Our current system implements standard agents and tools that enable optimization through Palimpzest. However, several enhancements could potentially lead to significantly improved performance:

- **Planner Agent.** While the Debugger and Code Editor Agents provide core functionality for bug localization and patch generation, the system currently lacks explicit planning between these stages. Introducing a Planner Agent to define structured steps following bug localization and prior to code editing could reduce the cognitive load on the Code Editor Agent and improve reasoning efficiency.

- **Validation Agent.** The system currently does not validate the correctness of generated code. Incorporating a Validation Agent capable of producing relevant test cases and executing them within the repository environment would serve as a critical guardrail.

This agent could detect erroneous patches early and provide actionable feedback to the Code Editor Agent to improve reliability.

- **Comprehensive Patch Correction.** A substantial number of instances fail due to incorrect patch formatting. Although the current system can test patches and correct basic issues such as indentation, line numbers, and file names, some formatting failures persist until the maximum iteration limit is reached. This suggests an opportunity to enhance tool feedback mechanisms to improve patch formatting earlier in the process and reduce the number of iterations required for successful correction.

- **Model Selection.** Exploring a broader range of high-performing models, such as Anthropic's Claude series, may lead to improved outcomes across both debugging and code generation tasks.

5.3 Future Work

While this work provides a solid foundation, several directions remain open for future exploration:

- **Dynamic Agent Plan Execution.** Many agentic frameworks support dynamic execution, which is characterized by non-deterministic communication between agents and the traversal of non-linear execution graphs. At present, Palimpzest is limited to executing linear plans, where nodes are evaluated in a predefined sequence. Extending Palimpzest to support non-linear plan execution would enable the optimization of more dynamic agentic workflows, unlocking a broader space of hyperparameters such as branching limits and tools that involve conditional routing. This extension would also facilitate the exploration of diverse agent architectures, ranging from coordinator-based designs in which a central agent delegates tasks to fully decentralized systems with distributed decision-making.

- **Agentic Logical Plan Optimizations.** Palimpzest is capable of generating multiple candidate logical plans for optimization, but this functionality currently exists only for non-agentic operators, such as filter. In workflows involving multiple agents, there is significant opportunity for plan level optimization, as different agent configurations and execution orders may lead to varying levels of performance. For example, if an agent is capable of generating and evaluating tests related to a given issue, executing this agent earlier in the plan may enable a test-first approach that provides valuable traces and feedback before patch generation. Supporting agent aware plan generation would thus enhance Palimpzest's ability to optimize more complex and adaptive agentic workflows.

- **Tool Optimizations.** Given the significant impact of tool design on agent performance, a promising direction for future hyperparameter optimization is the exploration of tool implementations and their associated parameters. For example, in the current implementation of the `get_file_content` tool, up to 200 lines of code are returned starting from a specified line number. However, varying this value may lead to improved agent performance. Returning more lines could reduce the number of iterations required to locate relevant code, whereas returning fewer lines may help conserve context window capacity. Optimizing such tool specific hyperparameters could lead to more efficient and effective agent behavior.

- **Full SWE-Bench Evaluation.** Due to computational constraints, we were unable to evaluate our system on the full SWE-Bench dataset. Future work may extend this evaluation to the complete benchmark, providing additional insights into system performance across a broader set of instances.

References

[1] C. Liu, M. Russo, M. Cafarella, L. Cao, P. B. Chen, Z. Chen, M. Franklin, T. Kraska, S. Madden, and G. Vitagliano. *A Declarative System for Optimizing AI Workloads*. 2024. arXiv: 2405.14696 [cs.CL].

[2] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. "SWE-bench: Can Language Models Resolve Real-world Github Issues?" In: *The Twelfth International Conference on Learning Representations*. 2024. URL: https://openreview.net/forum?id=VTF8yNQM66.

[3] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. R. Narasimhan, and O. Press. "SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering". In: (2024). URL: https://arxiv.org/abs/2405.15793.

[4] X. Wang et al. "OpenHands: An Open Platform for AI Software Developers as Generalist Agents". In: (2024). arXiv: 2407.16741 [cs.SE]. URL: https://arxiv.org/abs/2407.16741.

[5] M. A. Bel, J. L. Ríos, R. A. L. Carrasco, J. Michelini, G. Milano, G. Milano, M. Pérez, and G. Pasquero. "Globant Code Fixer Agent". In: *Globant Technical Report* (Nov. 2024). Contributors: Martin Alejandro Bel, José Lamas Ríos, Rodolfo Anibal Lobo Carrasco, Juan Michelini, Gastón Milano, German Milano, Marcelo Pérez, Guillermo Pasquero.

[6] C. Liu et al. "Palimpzest: Optimizing AI-Powered Analytics with Declarative Query Processing". In: *Proceedings of the Conference on Innovative Database Research (CIDR)*. 2025.

[7] M. Chen et al. "Evaluating Large Language Models Trained on Code". In: (2021). arXiv: 2107.03374 [cs.LG].

[8] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao. "ReAct: Synergizing Reasoning and Acting in Language Models". In: *arXiv preprint arXiv:2210.03629* (2022).

[9] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. "Chain-of-Thought Prompting Elicits Reasoning in Large Language Models". In: *arXiv preprint arXiv:2201.11903* (Jan. 2022). URL: https://arxiv.org/abs/2201.11903.

[10] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin. "CODEAGENT: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges". In: *Manuscript* (2024). Key Lab of High Confidence Software Technology (PKU), Ministry of Education. URL: https://example.com/codeagent-paper.

[11] H. N. Phan, P. X. Nguyen, and N. D. Bui. "HyperAgent: Generalist Software Engineering Agents to Solve Coding Tasks at Scale". In: *arXiv preprint arXiv:2406.11912* (2024).

[12] N. L. Hai, D. N. Manh, and N. D. Bui. "RepoExec: Evaluate Code Generation with a Repository-Level Executable Benchmark". In: *arXiv preprint arXiv:2406.11927v1* (2024).

[13] D. Chen et al. *CodeR: Issue Resolving with Multi-Agent and Task Graphs*. 2024. arXiv: 2406.01304 [cs.CL].