HyperAgent: Generalist Software Engineering Agents to Solve Coding Tasks at Scale

Huy Nhat Phan[1], **Phong X. Nguyen**[1] **and Nghi D. Q. Bui**[1,†]
[1]FPT Software AI Center, Viet Nam

Large Language Models (LLMs) have revolutionized software engineering (SE), demonstrating remarkable capabilities in various coding tasks. While recent efforts have produced autonomous software agents based on LLMs for end-to-end development tasks, these systems are typically designed for specific SE tasks. We introduce HYPERAGENT , a novel generalist multi-agent system designed to address a wide spectrum of SE tasks across different programming languages by mimicking human developers' workflows. Comprising four specialized agents—Planner, Navigator, Code Editor, and Executor—HYPERAGENT manages the full lifecycle of SE tasks, from initial conception to final verification. Through extensive evaluations, HYPERAGENT achieves state-of-the-art performance across diverse SE tasks: it attains a 25.01% success rate on SWE-Bench-Lite and 31.40% on SWE-Bench-Verified for GitHub issue resolution, surpassing existing methods. Furthermore, HYPERAGENT demonstrates superior performance in code generation at repository scale (RepoExec), and in fault localization and program repair (Defects4J), often outperforming specialized systems. This work represents a significant advancement towards versatile, autonomous agents capable of handling complex, multi-step SE tasks across various domains and languages, potentially transforming AI-assisted software development practices.

GitHub: https://github.com/FSoft-AI4Code/HyperAgent

1. Introduction

In recent years, Large Language Models (LLMs) have demonstrated remarkable capabilities in assisting with various coding tasks, ranging from code generation and completion to bug fixing and refactoring. These models have transformed the way developers interact with code, providing powerful tools that can understand and generate human-like code snippets with impressive accuracy. However, as software engineering tasks grow in complexity, there is an emerging need for more sophisticated solutions that can handle the intricacies of real-world software development.

To address these challenges, software agents built on top of LLMs have emerged as a promising solution. These agents are designed to automate and streamline complex software engineering tasks by leveraging the advanced reasoning and generative capabilities of LLMs. They can perform tasks such as code generation, bug localization, and even orchestrating multi-step development processes. Despite their potential, current software agents remain limited in scope. Most existing agents are designed to tackle a ***specific SE task*** with limited capability, such as resolving GitHub issues (Arora et al., 2024; Chen et al., 2024; Jimenez et al., 2023; Xia et al., 2024; Yang et al., 2024a; Zhang et al., 2024a) using benchmarks like SWE-bench (Jimenez et al., 2023). Others (Huang et al., 2023) focus on competitive code generation benchmarks, such as APPS (Hendrycks et al., 2021), HumanEval (Chen et al., 2021a), and MBPP (Austin et al., 2021). Another line of software agents (Hong et al., 2023; Nguyen et al., 2024; Qian et al., 2024) focuses on create complicated software given a set of requirements. While these specialized agents demonstrate impressive capabilities within their domains, the broader claim

Corresponding author(s): Nghi D. Q. Bui (bdqnghi@gmail.com)

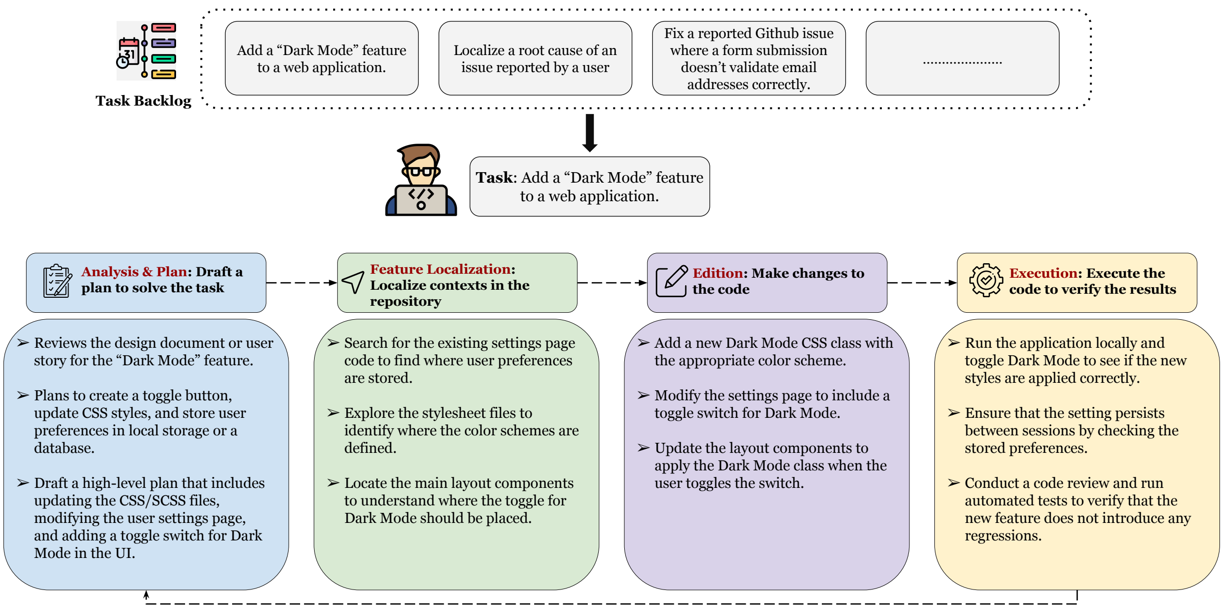

Task Backlog
Add a "Dark Mode" feature to a web application.
Localize a root cause of an issue reported by a user
Fix a reported Github issue where a form submission doesn't validate email addresses correctly.
....
Task: Add a "Dark Mode" feature to a web application.
Analysis & Plan: Draft a plan to solve the task
Feature Localization: Localize contexts in the repository
Edition: Make changes to the code
Execution: Execute the code to verify the results
➢ Reviews the design document or user story for the "Dark Mode" feature.
➢ Plans to create a toggle button, update CSS styles, and store user preferences in local storage or a database.
➢ Draft a high-level plan that includes updating the CSS/SCSS files, modifying the user settings page, and adding a toggle switch for Dark Mode in the UI.
➢ Search for the existing settings page code to find where user preferences are stored.
➢ Explore the stylesheet files to identify where the color schemes are defined.
➢ Locate the main layout components to understand where the toggle for Dark Mode should be placed.
➢ Add a new Dark Mode CSS class with the appropriate color scheme.
➢ Modify the settings page to include a toggle switch for Dark Mode.
➢ Update the layout components to apply the Dark Mode class when the user toggles the switch.
➢ Run the application locally and toggle Dark Mode to see if the new styles are applied correctly.
➢ Ensure that the setting persists between sessions by checking the stored preferences.
➢ Conduct a code review and run automated tests to verify that the new feature does not introduce any regressions.

Figure 1 | Illustration of a Developer's Workflow for Resolving a Software Engineering Task. The diagram outlines the key phases a developer typically follows when implementing a new feature, such as adding a "Dark Mode" to a web application. The workflow is divided into four main phases: Plan, where the developer drafts a solution strategy; Navigation, where the developer explores the codebase to identify relevant components; Edition, where the developer makes the necessary code changes; and Execution, where the developer tests and verifies the implementation. Each phase is represented with concrete steps to demonstrate how the task progresses from initial planning to final execution.

of solving general Software Engineering tasks may not fully materialize in practice. The diversity and interconnected nature of real-world SE challenges often require a more versatile approach that can seamlessly adapt to various tasks, programming languages, and development scenarios.

To address such drawbacks, we propose HyperAgent, a generalist multi-agent system designed to resolve a broad spectrum of SE tasks. Our design philosophy is rooted in the workflows that software engineers typically follow in their daily routines—whether it's implementing new features in an existing codebase, localizing bugs in a large project, or providing fixes for reported issues and so on. While developers may use different tools or approaches to tackle these tasks, they generally adhere to consistent workflow patterns. We illustrate this concept through a workflow that represents how developers typically resolve coding tasks. Although different SE tasks require varied approaches, they all follow a similar workflow. Figure 1 illustrates a typical workflow for a software engineer when resolving a task from the backlog, which is a list of tasks to be completed within a specific period.

1. **Analysis & Plan:** The developer starts by understanding the task requirements through documentation review and stakeholder discussions. A working plan is then formulated, outlining key steps, potential challenges, and expected outcomes. This plan remains flexible, adjusting as new insights are gained or challenges arise. For simple tasks, this plan may remain a mental checklist rather than a detailed written document.
2. **Feature Localization:** With a plan in place, the developer navigates the *repository* to identify relevant components, known as feature localization (Castro et al., 2019; Martinez et al., 2018; Michelon et al., 2021). This involves locating classes, functions, libraries, or modules pertinent to

the task. Understanding dependencies and the system's overall design is crucial here, ensuring the developer is ready to make informed modifications in the next phase.
3. **Edition:** The developer edits the identified code components, implementing changes or adding new functionality. This phase also involves ensuring smooth integration with the existing codebase, maintaining code quality, and adhering to best practices. New or updated unit tests are written to ensure functionality and reliability.
4. **Execution:** After editing, the developer tests the modified code to verify it meets the plan's requirements. This includes running unit and integration tests, as well as conducting manual testing or peer reviews. If issues are found, the process loops back to previous phases until the task is fully resolved.

These four steps are repeated iteratively until the developer confirms that the task is complete. Note that the exact steps may vary depending on the task and the developer's skill level. And the task may be completed in a single phase, or may require multiple iterations, i.e., after the Execution step, if the developer is not satisfied with the results, the developer may need to repeat the entire process. In HyperAgent, the framework is organized around four primary agents: *Planner*, *Navigator*, *Code Editor*, and *Executor*. An illustration of HyperAgentis depicted in Figure 2. Each agent corresponds to a specific step in the overall workflow depicted in Figure 1, though the actual workflow of each agent may differ slightly from how a human developer might approach similar tasks [1]. Our design emphasizes three main advantages over existing methods:

1. **Generalizability**: The framework is designed to easily adapt to a wide range of tasks with minimal configuration changes and little additional effort required to implement new modules into the system.
2. **Efficiency**: Each agent is optimized to manage processes with varying levels of complexity, requiring different degrees of intelligence from LLMs. For example, a lightweight and computationally efficient LLM can be employed for navigation, which, while less complex, involves the highest token consumption. Conversely, more complex tasks, such as code editing or execution, require more advanced LLM capabilities.
3. **Scalability**: The framework is built to scale effectively when deployed in real-world scenarios where the number of subtasks is significantly large. For instance, a complex task in the SWE-bench benchmark may require considerable time for an agent-based system to complete, and HyperAgentis designed to handle such scenarios efficiently.

We conducted extensive evaluations across diverse benchmarks to assess HyperAgent 's effectiveness in various software engineering tasks, demonstrating its versatility and superior performance. In GitHub issue resolution, HyperAgent outperformed strong baselines such as AutoCodeRover (Zhang et al., 2024c) and SWE-Agent (Yang et al., 2024a) on the SWE-bench benchmark (Jimenez et al., 2023), while for code generation at repository-level scale, it surpassed strong retrieval-augmented generation (RAG) baselines, including WizardLM2 (Luo et al., 2023) and GPT-3.5-Turbo on the RepoExec benchmark (Hai et al., 2024). In fault localization and program repair, HyperAgent achieved state-of-the-art performance on the Defects4J dataset (Sobreira et al., 2018), outperforming SOTA baselines such as DeepFL Li et al. (2019), AutoFL (Kang et al., 2024), RepairAgent (Bouzenia et al., 2024), and SelfAPR (Ye et al., 2022). Notably, *HyperAgent is the first system to evaluate SWE-Bench using open-source models such as Llama-3*, offering a more cost-effective solution compared to closed-source alternatives while maintaining competitive performance across a wide range of software engineering tasks.

[1] Details about each agent, along with how these advantages are achieved, are provided in Sections 5

These evaluations underscore HyperAgent 's unique position as the first system designed to work off-the-shelf across various SE tasks and programming languages, often exceeding the performance of specialized systems. This versatility highlights HyperAgent 's potential as a transformative tool in real-world software development scenarios, capable of adapting to diverse challenges without the need for task-specific fine-tuning. In summary, the key contributions of this work include:

- Introduction of HyperAgent , a novel generalist multi-agent system that closely mimics typical software engineering workflows. It incorporates stages for analysis, planning, feature localization, code editing, and execution/verification, enabling it to handle a broad spectrum of software engineering tasks across different programming languages.
- Extensive evaluation demonstrating superior performance across various software engineering benchmarks, including Github issue resolution (SWE-Bench-Python), repository-level code generation (RepoExec-Python), and fault localization and program repair (Defects4J-Java). To our knowledge, HyperAgent is the first system designed to work off-the-shelf across diverse SE tasks in multiple programming languages without task-specific adaptations.
- Insights into the design and implementation of scalable, efficient, and generalizable software engineering agent systems, paving the way for more versatile AI-assisted development tools that can seamlessly integrate into various stages of the software lifecycle.

2. Related Work

2.1. Deep Learning for Automated Programming

In recent years, applying deep learning to automated programming has captured significant interest within the research community (Allamanis et al., 2018; Balog et al., 2016; Bui and Jiang, 2018; Bui et al., 2021, 2023; Feng et al., 2020; Guo et al., 2020, 2022b; Wang et al., 2021). Specifically, Code Large Language Models (CodeLLMs) have emerged as a specialized branch of LLMs, fine-tuned for programming tasks (Allal et al., 2023; Bui et al., 2022; Feng et al., 2020; Guo et al., 2024; Li et al., 2023; Lozhkov et al., 2024; Luo et al., 2023; Nijkamp et al., 2022; Pinnaparaju et al., 2024; Roziere et al., 2023; Wang et al., 2021, 2023; Xu et al., 2022; Zheng et al., 2024). These models have become foundational in building AI-assisted tools for developers, aiming to solve competitive coding problems from benchmarks such as HumanEval (Chen et al., 2021b), MBPP (Austin et al., 2021), APPs (Hendrycks et al., 2021) and CRUXEval Gu et al. (2024a).

Despite achieving good results with benchmark tasks, these models often struggle to generate real-world software that requires complex logic and detailed acceptance criteria, which are essential for practical applications (Hong et al., 2024; Nguyen et al., 2024; Qian et al., 2023). This limitation has led to the development of more sophisticated benchmarks and evaluation methods.

2.2. Benchmarks for Software Engineering

Subsequent works have introduced new SE benchmarks that expand the scope and complexity of evaluation criteria. These efforts include translating problems across programming languages (Cassano et al., 2022; Wang et al., 2022), incorporating third-party libraries (Lai et al., 2023; Liu et al., 2023c), introducing derivative code completion tasks (Muennighoff et al., 2023), test coverage (Liu et al., 2023a), modifying edit scope (Ding et al., 2024; Du et al., 2023; Yu et al., 2024), and enhancing robustness to dataset contamination (Naman Jain et al., 2024). However, these code generation problems remain largely self-contained, with short problem descriptions (~100 lines) and correspondingly brief solutions, typically requiring only basic language primitives. As language models (LMs) rapidly evolve, many of these benchmarks are becoming saturated, highlighting the need for more realistic and challenging tasks that demand complex reasoning and problem-solving skills.

Real-world software engineering often requires understanding the broader context of a problem, identifying relevant parts of a repository, and generating code that meets specific acceptance criteria. This complexity has led to the development of repository-level code generation benchmarks (Hai et al., 2024; Li et al., 2024; Liu et al., 2023b; Zhang et al., 2023). Furthermore, recognizing that software engineering encompasses diverse tasks beyond code generation, researchers have developed benchmarks to address various aspects of the field. SWE-bench (Jimenez et al., 2023) mimics the workflow of resolving GitHub issues, while Defects4J (Just et al., 2014) and BugsInPy (Widyasari et al., 2020) are popular benchmarks for fault localization and program repair. CodeXGlue (Lu et al., 2021) offers a comprehensive benchmark covering multiple software engineering tasks. These developments reflect the ongoing efforts to create more holistic and challenging evaluations that better represent the complexities of real-world software development.

2.3. Autonomous Coding Agents

The emergence of open-source software development tools based on large language models (LLMs) has revolutionized the field of autonomous coding. These tools leverage LLMs' capabilities to plan, self-critique, and extend functionality through function calls. By integrating such tools into development workflows, researchers have observed dramatic performance improvements in code generation tasks on popular benchmarks such as HumanEval (Chen et al., 2021b). Notable advancements in this area include works by Huang et al. (2023), Chen et al. (2023), Shinn et al. (2024), Islam et al. (2024), Chen et al. (2022), and To et al. (2024). A parallel line of research focuses on generating complex software systems comprising multiple executable code files from input software requirements. Significant contributions in this domain include MetaGPT (Hong et al., 2023), AgileCoder (Nguyen et al., 2024), and ChatDev (Qian et al., 2024). These approaches aim to automate larger portions of the software development process, moving beyond single-file code generation.

Recently, there has been growing interest in employing coding agents to automatically resolve GitHub issues, a task that more closely mimics real-world software engineering challenges. This trend is evident in works such as SWE-Agent (Yang et al., 2024a), SWE-bench (Jimenez et al., 2023), AutoCodeRover (Zhang et al., 2024c), and agentless approaches (Xia et al., 2024). The integration of interaction and code generation has spawned novel applications where code serves as the primary modality for actions (Wang et al., 2024a; Yang et al., 2024b), tool construction (Gu et al., 2024b; Wang et al., 2024b; Zhang et al., 2024b), and reasoning (Shinn et al., 2024; Zelikman et al., 2023a,b). Beyond general software engineering tasks, code language agents have found applications in specialized domains such as offensive security (Shao et al., 2024; Yang et al., 2023) and theorem proving (Thakur et al., 2023). This evolution towards agent-based models represents a significant step in bridging the gap between academic benchmarks and real-world software engineering challenges. By mimicking human-like problem-solving processes in coding tasks, these autonomous agents are paving the way for more sophisticated and practical AI-assisted development tools, potentially transforming the landscape of software engineering.

3. Problem Formulation

To formally define the software engineering tasks that HyperAgent is designed to address, we introduce the following notation and definitions:

Software Engineering Task Let $\mathcal{R} = \{r_1, r_2, ..., r_n\}$ be a software repository consisting of n code files. Each file r_i is a sequence of code tokens. A Software Engineering (SE) task T is defined as a tuple $T = (D, \mathcal{R}, F)$, where:

- D is a natural language description of the task requirements
- $\mathcal{R}$ is the software repository on which the task is to be performed
- $F : \mathcal{R} \rightarrow \mathcal{R}'$ is the desired transformation function that modifies the repository to fulfill the task requirements

The goal of an SE agent is to approximate the function F given D and $\mathcal{R}$, producing a modified repository $\mathcal{R}'$ that satisfies the task requirements. We can further categorize SE tasks into several types based on their specific objectives.

Issue Resolution Task An Issue Resolution Task $T_{IR} = (I, \mathcal{R}, F_{IR})$ is an SE task where I is a description of a GitHub issue, and F_{IR} is a function that modifies $\mathcal{R}$ to resolve the issue.

Code Generation Task A Code Generation Task $T_{CG} = (S, \mathcal{R}, F_{CG})$ is an SE task where S is a specification for new code to be generated, and F_{CG} is a function that adds new code to $\mathcal{R}$ according to S.

Fault Localization Task A Fault Localization Task $T_{FL} = (B, \mathcal{R}, F_{FL})$ is an SE task where B is a description of a bug or failing test case, and $F_{FL} : \mathcal{R} \rightarrow \mathcal{L}$ is a function that identifies a set of locations $\mathcal{L}$ in $\mathcal{R}$ where the bug is likely to be present.

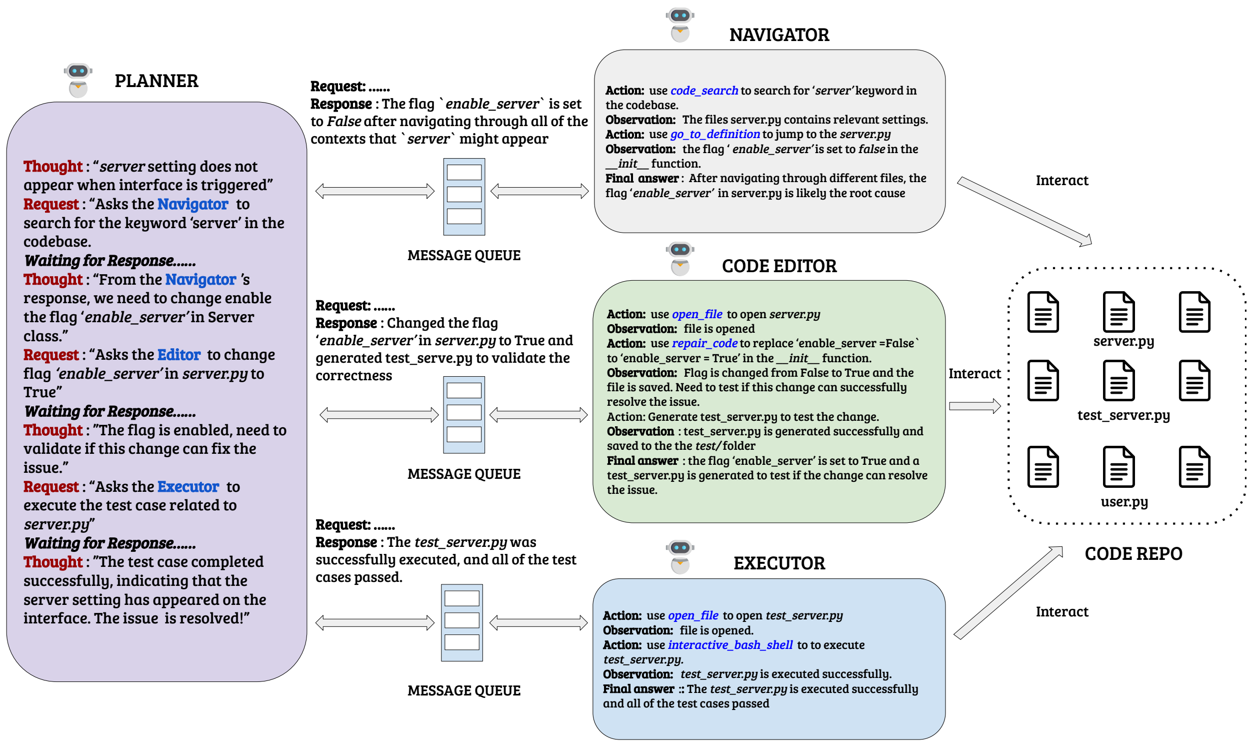
Program Repair Task A Program Repair Task $T_{PR} = (B, \mathcal{R}, F_{PR})$ is an SE task where B is a description of a bug or failing test case, and F_{PR} is a function that modifies $\mathcal{R}$ to fix the bug while preserving correct functionality.

Generalist Software Engineering Agent Given a set of diverse SE tasks $\mathcal{T} = \{T_1, T_2, ..., T_m\}$, where each T_i can be any type of SE task (e.g., T_{IR}, T_{CG}, T_{FL}, or T_{PR}), design an agent A that can effectively perform all tasks in $\mathcal{T}$ by approximating the corresponding transformation functions F_i for each task T_i. The challenge in designing such a generalist agent lies in creating a unified framework that can: (1) Understand and interpret diverse task descriptions D_i; (2) Navigate and comprehend different repository structures $\mathcal{R}_i$; (3) Generate appropriate code modifications or analysis results for each task type; (4) Verify the correctness and effectiveness of the solutions.

HyperAgent addresses this challenge by employing a multi-agent architecture that mimics the workflow of human developers. Each agent in the system specializes in a specific aspect of the software engineering process, allowing for a modular and adaptable approach to diverse SE tasks.

4. HyperAgent: A Generalist Software Agent Framework

Figure 2 illustrates the HyperAgent framework. The key design principle of HyperAgent is the centralization of advanced reasoning in the *Planner* agent, with delegation of computationally intensive but conceptually simpler tasks to specialized child agents. This approach optimizes inference costs and overall performance by eliminating redundant information processing outside the *Planner's* context.



Thought : "*server* setting does not appear when interface is triggered"
Request : "Asks the **Navigator** to search for the keyword 'server' in the codebase.
Waiting for Response......
Thought : "From the **Navigator** 's response, we need to change enable the flag '*enable_server*' in Server class."
Request : "Asks the **Editor** to change flag '*enable_server*' in *server.py* to True"
Waiting for Response......
Thought : "The flag is enabled, need to validate if this change can fix the issue."
Request : "Asks the **Executor** to execute the test case related to *server.py*"
Waiting for Response......
Thought : "The test case completed successfully, indicating that the server setting has appeared on the interface. The issue is resolved!"

Request: ......
Response : The flag \`*enable_server*\` is set to *False* after navigating through all of the contexts that \`*server*\` might appear

Request: ......
Response : Changed the flag '*enable_server*' in *server.py* to True and generated test_serve.py to validate the correctness

Request: ......
Response : The *test_server.py* was successfully executed, and all of the test cases passed.

Action: use *code_search* to search for '*server*' keyword in the codebase.
Observation: The files server.py contains relevant settings.
Action: use *go_to_definition* to jump to the *server.py*
Observation: the flag ' *enable_server*' is set to *false* in the *__init__* function.
Final answer : After navigating through different files, the flag '*enable_server*' in server.py is likely the root cause

Action: use *open_file* to open *server.py*
Observation: file is opened
Action: use *repair_code* to replace 'enable_server =False\` to 'enable_server = True' in the *__init__* function.
Observation: Flag is changed from False to True and the file is saved. Need to test if this change can successfully resolve the issue.
Action: Generate test_server.py to test the change.
Observation : test_server.py is generated successfully and saved to the the *test/* folder
Final answer : the flag 'enable_server' is set to True and a test_server.py is generated to test if the change can resolve the issue.

Action: use *open_file* to open *test_server.py*
Observation: file is opened.
Action: use *interactive_bash_shell* to to execute *test_server.py.*
Observation: *test_server.py* is executed successfully.
Final answer :: The *test_server.py* is executed successfully and all of the test cases passed

Figure 2 | Overview of HyperAgent: A scalable, multi-agent system for software engineering tasks. The workflow illustrates the central *Planner* agent coordinating with specialized child agents (*Navigator*, *Editor*, and *Executor*) through an asynchronous Message Queue. This architecture enables parallel processing of subtasks, dynamic load balancing, and efficient handling of complex software engineering challenges.

4.1. Centralized Multi-Agent System

The HyperAgent framework comprises four primary agents:

- **Planner**: The *Planner* agent serves as the central decision-making unit. It processes human task prompts, generates resolution strategies, and coordinates child agent activities. The *Planner* operates iteratively, generating plans, delegating subtasks, and processing feedback until task completion or a predefined iteration limit is reached.
- **Navigator**: The *Navigator* agent specializes in efficient information retrieval within the codebase. Equipped with IDE-like tools such as `go_to_definition` and `code_search`, it traverses codebases rapidly, addressing challenges associated with private or unfamiliar code repositories. The *Navigator* is designed for speed and lightweight operation, utilizing a combination of simple tools to yield comprehensive search results.
- **Editor**: The *Editor* agent is responsible for code modification and generation across multiple files. It employs tools including `auto_repair_editor`, `code_search`, and `open_file`. Upon receiving target file and context information from the *Planner*, the *Editor* generates code patches, which are then applied using the `auto_repair_editor`.
- **Executor**: The *Executor* agent validates solutions and reproduces reported issues. It utilizes an `interactive_bash_shell` for maintaining execution states and `open_file` for accessing relevant documentation. The *Executor* manages environment setup autonomously, facilitating efficient testing and validation processes.

4.2. Agent Communication and Scalability

Inter-agent communication in HyperAgent is structured to minimize information loss, ensure effective task delegation, and enable scalable, parallel processing of complex software engineering tasks. To achieve these goals, we implement an asynchronous communication model using a distributed Message Queue system based on Redis.

The *Planner* communicates with child agents using a standardized message format comprising Context and Request fields. The Context field provides relevant background information and rationale for the requested action, while the Request field contains specific, actionable instructions for the child agent. This structure enables the *Planner* to convey both high-level context and specific directives efficiently.

When the *Planner* needs to delegate a task, it decomposes it into subtasks and publishes messages containing these subtasks to appropriate queues in the Message Queue system. Child agents, including multiple instances of the *Navigator*, *Editor*, and *Executor*, continuously monitor these queues and process tasks asynchronously. This approach allows for parallel processing of subtasks, significantly improving the system's efficiency and scalability.

For instance, when exploring a large codebase, multiple *Navigator* instances can simultaneously investigate different sections, dramatically reducing exploration time. Similarly, the *Editor* can parallelize large-scale code changes across multiple files, and the *Executor* can run multiple tests concurrently, greatly accelerating the validation process.

To mitigate information loss in child agent reports, we implement a lightweight *LLM summarizer*. This component compiles intermediate results from the child agent's execution log, generating a concise yet comprehensive summary. Upon task completion, child agents publish these summarized results back to a designated queue for the *Planner* to aggregate and process. This approach preserves critical details about code snippets, explored objects, and codebase structure, reducing the risk of information degradation or hallucination in the *Planner* over multiple iterations. The Message Queue-based architecture offers several advantages:

- **Parallel Processing**: Multiple instances of each agent type can work on different subtasks simultaneously, significantly improving overall system throughput.
- **Load Balancing**: Tasks can be dynamically distributed among multiple instances of each agent type, allowing for efficient resource utilization.
- **Fault Tolerance**: If an agent instance fails, unprocessed tasks remain in the queue and can be redistributed to other available instances, enhancing system reliability.
- **Scalability**: The system can easily scale horizontally by adding more instances of child agents to handle increased workload, without modifying the core architecture.
- **Decoupling**: The Message Queue decouples the *Planner* from the child agents, allowing for independent scaling and maintenance of each component.

This scalable, asynchronous communication model enables HyperAgent to efficiently handle complex software engineering tasks in large-scale, distributed environments. It adapts dynamically to varying workloads and task complexities, making it well-suited for real-world software development scenarios where task volume and complexity can fluctuate significantly.

4.3. Tool Design

The efficacy of HyperAgent is significantly enhanced by its specialized tool design. Key considerations in tool development include feedback format, functionality, and usability. Tools provide succinct, informative, and LLM-interpretable output. Each tool is optimized for its specific role in the software engineering process. Input interfaces are designed for intuitive interaction with LLMs, minimizing the risk of incorrect usage.

The *Navigator* employs a suite of navigation tools. The `code_search` tool utilizes a trigram-based search engine (Zoekt) [2] with symbol ranking. IDE-inspired features such as `go_to_definition`, `get_all_references`, and `get_all_symbols` enable precise code navigation. The `get_tree_structure` tool visualizes codebase structure, while `open_file` displays source code with integrated keyword search functionality. A proximity search algorithm is implemented to address LLM limitations in providing precise positional inputs.

The *Editor* utilizes the `repair_editor` tool, which applies and refines code patches, automatically addressing common syntax and indentation issues. It also employs navigation tools for context-aware editing.

The *Executor* employs an `interactive_shell` that maintains execution states for command sequences. It also uses `open_file` and `get_tree_structure` to facilitate access to testing and setup documentation.

5. Implementation Details

For *summarizer* in HyperAgent , we used Mixtral 8x7B (Jiang et al., 2024). To examine the flexibility of our framework and measure robustness, we employed a variety of language models (LMs) across different configurations. We tested four main configurations of HyperAgent , each utilizing different combinations of LMs for the Planner, Navigator, Editor, and Executor roles. Table 3 presents these configurations.

An advantage of our design is that we can choose the most suitable LLMs for each agent type, optimizing performance and accuracy. For instance, the *Planner*, serving as the brain of the whole system, requires a powerful model with superior reasoning capabilities to effectively orchestrate complex tasks. The *Editor* also demands a strong model with robust coding capability to edit and generate code accurately, given the inherent complexity of real-world codebases. In contrast, the *Navigator* and *Executor* agents can utilize less powerful models with smaller footprints and faster inference times, as their tasks are more straightforward and require less complex reasoning. This flexible architecture enables efficient allocation of computational resources, ensuring optimal performance across different agent types while balancing the trade-offs between model capability and computational cost. Such a design also allows for easier updates and improvements to individual components without necessitating a complete system overhaul.

As a result, we can implement various configurations of HyperAgent as shown in Table 3, utilizing both open-source and closed-source models. For closed-source models, we designate GPT-4 and Claude-3 Sonnet as strong models, while Claude-3 Haiku serves as the weak model. In the open-source domain, Llama-3-70B acts as the strong model, with Llama-3-8B serving as the weak model. We believe that *HyperAgent is the first system to evaluate SWE-Bench using open-source models such as Llama-3*, offering a more cost-effective solution compared to closed-source alternatives while maintaining competitive performance across a wide range of software engineering tasks.

6. Evaluations

We conducted extensive evaluations of HyperAgent across diverse benchmarks to assess its effectiveness in various software engineering tasks. Our criteria for selecting SE tasks and corresponding

[2] https://github.com/google/zoekt

Table 1 | HyperAgent : Specialized Tool Design by Agent

Agent	Tool	Description
Navigator	`code_search`	Trigram-based search engine (Zoekt) with symbol ranking
	`go_to_definition`	Locates and displays the definition of a given symbol
	`get_all_refs`	Finds all references to a specific symbol in the codebase
	`get_all_symbols`	Lists all symbols (functions, classes, etc.) in a given file or module
	`get_tree_struc`	Visualizes the codebase structure as a tree
	`open_file`	Displays source code with integrated keyword search functionality
Editor	`repair_editor`	Applies and refines code patches, addressing syntax and indentation issues
	Navigation tools	Employs Navigator's tools for context-aware editing
Executor	`interactive_shell`	Maintains execution states for command sequences
	`open_file`	Accesses testing and setup documentation
	`get_tree_struc`	Visualizes structure of test suites and configuration files

Table 2 | HyperAgent Specialized Tool Design: A comprehensive overview of the custom-designed tools for each agent type (Navigator, Editor, and Executor). These tools are optimized for efficient code exploration, precise editing, and robust execution, enabling HyperAgent to handle complex software engineering tasks with high accuracy and performance. The specialized nature of these tools, coupled with their LLM-friendly interfaces, allows for seamless integration within the multi-agent system, facilitating effective collaboration between agents and enhancing overall system capabilities.

benchmarks were based on complexity and real-world relevance. Each task must require multiple reasoning steps to complete, such as retrieving relevant contexts from the repository, editing code, and executing tests. We focused on well-known tasks including GitHub issue resolution, code generation at repository-level scale, fault localization, and program repair.

- **GitHub Issue Resolution:** This task requires complex reasoning steps to understand the issue description, explore the code repository and identify the relvant contexts, edit the code and evaluate if the issues have been resolved.
- **Code Generation at Repository-Level Scale:** While code generation is a popular task for evaluating AI systems' coding capabilities, many benchmarks use competitive programming tasks (Austin et al., 2021; Chen et al., 2021a; Hendrycks et al., 2021) that don't reflect real-world software development scenarios. Instead, we choose benchmarks that evaluate code generation at the repository level, requiring the system to retrieve relevant contexts, understand the codebase, and generate code consistent with existing structures.

Table 3 | HyperAgent Configurations

Configuration	Planner	Navigator	Editor	Executor
HyperAgent-Lite-1	Claude-3-Sonnet	Claude-3-Haiku	Claude-3-Sonnet	Claude-3-Haiku
HyperAgent-Lite-2	Llama-3-70B	Llama-3-8b	Llama-3-70B	Llama-3-8b
HyperAgent-Full-1	Claude-3-Sonnet	Claude-3-Sonnet	Claude-3-Sonnet	Claude-3-Sonnet
HyperAgent-Full-2	GPT-4o	GPT-4o	GPT-4o	GPT-4o
HyperAgent-Full-3	Llama-3-70B	Llama3-70B	Llama-3-70B	Llama-3-70B

- **Fault Localization:** This task involves identifying the specific location of a bug within a program. It requires analyzing program behavior, understanding code structure, and pinpointing the exact lines or components responsible for faulty behavior. Our benchmark assesses the system's ability to accurately locate bugs in complex codebases, simulating real-world debugging scenarios.
- **Program Repair:** Building upon fault localization, program repair involves automatically fixing identified bugs. This task challenges the system to not only locate the bug but also generate appropriate patches. Our benchmark evaluates the quality and correctness of proposed fixes, considering factors such as maintaining original program functionality and adhering to coding standards.

6.1. GitHub Issue Resolution

6.1.1. Dataset

We evaluated HyperAgent using the SWE-bench benchmark (Jimenez et al., 2023), which comprises 2,294 task instances derived from 12 popular Python repositories. SWE-bench assesses a system's capability to automatically resolve GitHub issues using Issue-Pull Request (PR) pairs, with evaluation based on verifying unit tests against the post-PR behavior as the reference solution. Due to the original benchmark's size and the presence of underspecified issue descriptions, we utilized two refined versions: SWE-bench-Lite (300 instances) and SWE-bench-Verified (500 instances). The Lite version filters samples through heuristics (e.g., removing instances with images, external hyperlinks, or short descriptions), while the Verified version contains samples manually validated by professional annotators. These streamlined versions offer a more focused and reliable evaluation framework, addressing the limitations of the original benchmark while maintaining its core objectives.

6.1.2. Baselines

We compared HyperAgent to several strong baselines: SWE-Agent (Yang et al., 2024a), a bash interactive agent with Agent-Computer Interfaces; AutoCodeRover (Zhang et al., 2024c), a two-stage agent pipeline focusing on bug fixing scenarios; Agentless (Xia et al., 2024), a simplified two-phase approach that outperforms complex agent-based systems in software development tasks; and various Retrieval Augmented Generation (RAG) baselines as presented in (Jimenez et al., 2023). These baselines represent a diverse range of approaches to software engineering tasks, providing a comprehensive evaluation framework for our method.

6.1.3. Metrics

We evaluate this task using three key metrics: (1) percentage of resolved instances, (2) average time cost, and (3) average token cost. The percentage of resolved instances measures overall effectiveness, indicating the proportion of SWE-bench tasks where the model generates solutions passing all unit tests, thus fixing the described GitHub issue. Average time cost assesses efficiency in processing and resolving issues, while average token cost quantifies economic efficacy through computational resource usage. These metrics collectively provide a comprehensive evaluation of each tool's performance in addressing real-world software problems, balancing success rate with time and resource utilization.

6.1.4. Results

Method	Verified (%)	Lite (%)	Avg Time	Avg Cost ($)
AutoCodeRover + GPT-4o	28.80	22.7	720	0.68
SWE-Agent + Claude 3.5 Sonnet	33.60	23.00	–	1.79
SWE-Agent + GPT-4o	23.20	18.33	–	2.55
Agentless + GPT-4o	33.20	24.30	–	0.34
RAG + Claude 3 Opus	7.00	4.33	–	–
HyperAgent-Lite-1	27.33	21.67	132	0.45
HyperAgent-Lite-2	16.00	11.00	108	0.76
HyperAgent-Full-1	31.00	24.67	320	1.82
HyperAgent-Full-2	**31.40**[*]	**25.00**	210	2.01
HyperAgent-Full-3	–	–	–	–

[*] Best performance among all methods

Table 4 | Performance comparison on SWE-Bench datasets. Verified (%) and Lite (%) columns show the percentage of resolved instances (out of 500 for Verified, 300 for Lite). Avg Time is in seconds, and Avg Cost is in US dollars.

The results presented in Table 4 demonstrate the competitive performance of HyperAgentacross different configurations on the SWE-Bench datasets. Several key observations can be made:

1. **Performance:** HyperAgent-Full-2 achieves a strong success rate of 31.40% on the SWE-Bench Verified dataset. This performance is competitive with top-performing methods such as SWE-Agent + Claude 3.5 Sonnet (33.60%) and Agentless + GPT-4o (33.20%). On the SWE-Bench Lite dataset, HyperAgent-Full-2 achieves the best performance among all methods with a 25.00% success rate, closely followed by HyperAgent-Full-1 at 24.67%. This outperforms strong baselines like Agentless + GPT-4o (24.30%) and SWE-Agent + Claude 3.5 Sonnet (23.00%).
2. **Efficiency:** HyperAgent-Lite configurations demonstrate impressive efficiency. HyperAgent-Lite-1 and HyperAgent-Lite-2 have average processing times of 132 and 108 seconds respectively, significantly faster than AutoCodeRover + GPT-4o (720 seconds).
3. **Cost-Effectiveness:** HyperAgent-Lite-1 offers an excellent balance of performance (27.33% on Verified, 21.67% on Lite) and cost ($0.45). This makes it substantially more cost-effective than several baselines, including SWE-Agent + Claude 3.5 Sonnet ($1.79) and SWE-Agent + GPT-4o ($2.55).
4. **Scalability:** The performance spectrum across HyperAgent-Lite and HyperAgent-Full configurations demonstrates the system's adaptability to different performance-cost trade-offs, providing flexibility for various use cases.

Overall, HyperAgentdemonstrates strong and competitive performance across both datasets. HyperAgent-Full-2 achieves the best performance on the Lite dataset and is highly competitive on the Verified dataset. The system's ability to achieve high success rates while offering various configurations for efficiency and cost-effectiveness positions HyperAgentas a versatile and practical solution for automated GitHub issue resolution. Its performance is particularly noteworthy given its use of open-source models, offering a compelling alternative to systems relying on closed-source models.

6.2. Repository-Level Code Generation

6.2.1. *Dataset*

We evaluate our task using RepoExec (Hai et al., 2024), a benchmark for Python for assessing repository-level code generation with emphasis on executability and correctness. Comprising 355 samples with automatically generated test cases (96.25% coverage), RepoExec typically provides gold contexts extracted through static analysis. The gold contexts are splitted into different richness level, including full context, medium context and small context. The richness level of contexts represent for different way to retrieve the contexts, such as import, docstring, function signature, API invocaction, etc. However, to measure HyperAgent's ability to navigate codebases and extract contexts independently, we omit these provided contexts in our evaluation.

6.2.2. *Baselines*

We compared HyperAgent against strong retrieval-augmented generation (RAG) baselines, including WizardLM2 + RAG, GPT-3.5-Turbo + RAG, WizardLM2 + Sparse RAG, and GPT-3.5-Turbo + Sparse RAG. These baselines represent state-of-the-art approaches in combining large language models with information retrieval techniques. Sparse RAG represents for using BM25 retriever and RAG stands for using UnixCoder Guo et al. (2022a) as context retriever. We used chunking size of 600 and python code parser from Langchain [3] allowing us to parse the context in a syntax-aware manner. Additionally, we included results from CodeLlama (34b and 13b versions) and StarCoder models when provided with full context from RepoExec, serving as upper bounds for performance with complete information.

6.2.3. *Metrics*

We used pass@1 and pass@5 as our primary metric, which measures the percentage of instances where all tests pass successfully after applying the model-generated patch to the repository.

Model	**Context Used**	**Pass@1**	**Pass@5**	**Cost ($)**
CodeLlama-34b-Python	Full	**42.93%**	49.54%	–
CodeLlama-13b-Python	Full	38.65%	43.24%	–
StarCoder	Full	28.08%	33.95%	–
WizardLM2 + RAG	Auto-retrieved	33.00%	49.16%	0.04
GPT-3.5-Turbo + RAG	Auto-retrieved	24.16%	35.00%	0.02
WizardLM2 + Sparse RAG	Auto-retrieved	34.16%	51.23%	0.05
GPT-3.5-Turbo + Sparse RAG	Auto-retrieved	25.00%	35.16%	0.03
HyperAgent-Lite-3	Auto-retrieved	38.33%	**53.33%**	0.18

Table 5 | RepoExec Results Comparison: HyperAgent-Lite-3 achieves comparable or superior performance to models provided with full context, particularly in Pass@5 (53.33%). It outperforms RAG-based models, demonstrating effective automatic context retrieval from codebases. This highlights the potential of end-to-end solutions like HyperAgentin real-world scenarios where manual context provision is impractical.

[3]https://github.com/langchain-ai/langchain

As shown in Table 5, the RepoExec benchmark results reveal insightful comparisons between different code generation approaches. CodeLlama-34b-Python, given full context, achieves the highest Pass@1 rate at 42.93%. Notably, our HyperAgent-Lite-3, which automatically retrieves relevant contexts, outperforms all models in Pass@5 at 53.33%, demonstrating its effective codebase navigation. In contrast, RAG-based models show limited effectiveness in capturing complex code relationships, underperforming both HyperAgentand full-context models. These findings highlight the potential of end-to-end solutions like HyperAgentfor real-world scenarios where manual context provision is impractical, emphasizing the need for sophisticated context retrieval methods in code generation tasks.

6.3. Fault Localization

6.3.1. Dataset

We evaluated HyperAgent on the Defects4J dataset (Just et al., 2014; Sobreira et al., 2018), a widely used benchmark for fault localization and program repair tasks. Our evaluation encompassed all 353 active bugs from Defects4J v1.0.

6.3.2. Baselines

We compared HyperAgent against several strong baselines, including DeepFL Li et al. (2019), AutoFL (Kang et al., 2024), Grace (Lou et al., 2021) DStar (Wong et al., 2012), and Ochiai (Zou et al., 2019). DeepFL, AutoFL and Grace represent more recent approaches that leverage deep learning methods for fault localization. In contrast, DStar and Ochiai are traditional techniques that employ static analysis-based methods to identify faults.

6.3.3. Metrics

We follow AutoFL (Kang et al., 2024) to use acc@k metric which measures the We adopt the acc@k metric from AutoFL to evaluate bug localization performance. This metric measures the number of bugs for which the actual buggy location is within a tool's top k suggestions. We choose this metric because previous research indicates that developers typically examine only a few suggested locations when debugging, and it's widely used in prior work. To handle ties in the ranking, we employ the ordinal tiebreaker method instead of the average tiebreaker, as we believe it more accurately reflects a developer's experience when using a fault localization tool.

6.3.4. Results

The fault localization results in Table 6 on the Defects4J dataset demonstrate HyperAgent superior performance, achieving an Acc@1 of 59.70%. This significantly outperforms all other methods, surpassing the next best performer, AutoFL, by 8.7 percentage points (51.00%) and more than doubling the accuracy of traditional methods like Ochiai (20.25%). HyperAgent's ability to correctly identify the buggy location on its first attempt for nearly 60% of the bugs suggests a potentially substantial reduction in debugging time and effort in real-world scenarios. The wide performance range across methods (20.25% to 59.70%) highlights both the challenges in fault localization and the significant improvement HyperAgentrepresents. While there's still room for improvement, these results indicate that HyperAgent's approach is more effective than existing deep learning and traditional static analysis-based methods for fault localization in Java projects.

Method	Acc@1	Cost ($)
Ochiai (Zou et al., 2019)	20.25%	–
DeepFL (Li et al., 2019)	33.90%	–
Dstar (Wong et al., 2012)	33.90%	–
Grace (Zou et al., 2019)	49.36%	–
AutoFL (Kang et al., 2024)	51.00%	–
HyperAgent-Lite-1	**59.70%**	0.18

Table 6 | Comparison of Acc@1 across Different Fault Localization Methods on the Defects4J dataset. HyperAgent-Lite-1 significantly outperforms all baselines, achieving 59.70% accuracy on this widely-used benchmark. It surpasses the next best method, AutoFL, by 8.7 percentage points, and more than doubles the performance of traditional methods like Dstar and Ochiai. This demonstrates the effectiveness of HyperAgent's approach in precisely locating faults on the first attempt in real-world Java projects, potentially reducing debugging time and effort for developers.

6.4. Program Repair

6.4.1. *Dataset*

We also utilize the Defects4J dataset (Just et al., 2014; Sobreira et al., 2018). This dataset is particularly suitable as it provides gold-standard fixes and test cases, which are crucial for evaluating the effectiveness of repair techniques once faults are localized and fixes are applied.

6.4.2. *Baselines*

We compared HyperAgent with configuration Lite-1 against state-of-the-art baselines: RepairAgent (Bouzenia et al., 2024), SelfAPR (Ye et al., 2022), and ITER (Ye and Monperrus, 2024). ITER and SelfAPR are learning-based methods, while RepairAgent is a multi-agent system leveraging LLMs to autonomously plan and execute bug fixes. RepairAgent interleaves information gathering, repair ingredient collection, and fix validation, dynamically selecting tools based on gathered information and previous fix attempts.

6.4.3. *Metrics*

As in previous studies Bouzenia et al. (2024); Hidvégi et al. (2024), we provide both the count of plausible and correct patches. A fix is considered plausible if it passes all the test cases, but this doesn't guarantee its correctness. To assess if a fix is correct, we automatically verify if its syntax aligns with the fix created by the developer via exactly matching Abstract Syntax Tree (AST) between fixes.

6.4.4. *Results*

The results presented in Table 7 on the Defects4J dataset demonstrate HyperAgent's strong performance compared to existing repair tools. HyperAgentachieved 192 correct fixes out of 835 bugs, outperforming its closest competitors: RepairAgent (164 correct fixes) and SelfAPR (110 correct fixes). Additionally, HyperAgentgenerated 249 plausible fixes, indicating its capability to produce a high number of test-passing patches. The performance metrics of HyperAgentare noteworthy, with 249 plausible fixes (29.8%) and 192 correct fixes (23%) out of 835 bugs. These results can be attributed to HyperAgent's four-phase repair strategy: planning, localization, fix verification via testing, and implementation. This approach enables HyperAgentto address the complex challenges of automated program repair effectively. HyperAgent's performance extends to the expanded Defects4Jv2 dataset, where it correctly fixed 110 out of 440 bugs. This demonstrates HyperAgent's ability to maintain high performance across a broader range of modern Java projects, highlighting its versatility and scalability in automated program repair tasks.

Project	**Bugs**	**HyperAgent**		**RepairAgent**	**ITER**	**SelfAPR**
		Plausible	**Correct**	**Correct**	**Correct**	**Correct**
Chart	26	20	14	11	10	7
Cli	39	18	10	8	6	8
Closure	174	30	24	27	18	20
Codec	18	12	9	9	3	8
Collections	4	1	1	1	0	1
Compress	47	12	9	10	4	7
Csv	16	8	7	6	2	1
Gson	18	5	4	3	0	1
JacksonCore	26	6	6	5	3	3
Jacksondatabind	112	21	14	11	0	8
JacksonXml	6	1	1	1	0	1
Jsoup	93	26	24	18	0	6
JxPath	22	3	2	0	0	1
Lang	63	24	19	17	0	10
Math	106	36	32	29	0	22
Mockito	38	20	12	6	0	3
Time	26	6	4	2	2	3
Defects4Jv1.2	395	119	82	74	57	64
Defects4Jv2	440	130	110	90	–	46
Total	**835**	**249**	**192**	**164**	**57**	**110**
Percentage		**(29.8%)**	**(22.9%)**	**(19.64%)**	**(6.82%)**	**(13.17%)**

Table 7 | Results on Defects4J dataset comparing HyperAgentwith other repair tools. The table includes the number of bugs, and for HyperAgent, both plausible and correct fixes. For RepairAgent, ITER, and SelfAPR, only the number of correct fixes is shown. Note that ITER does not have results for Defects4Jv2. HyperAgentachieves the best performance with 249 plausible fixes and 192 correct fixes (highlighted in blue).

7. Analysis

7.1. Ablation Studies on Agent Roles

We conducted experiments using SWE-bench Tiny to evaluate the contribution of each agent role to overall performance. This was done by replacing each child agent with the planner itself, requiring the planner to directly utilize the eliminated agent's toolset. Table 8 illustrates a significant cost increase for all configurations when any agent role is removed. The resolving rate also decreases, with the magnitude varying based on which role is eliminated. Removing the *Navigator* causes the most substantial performance drop, followed by the *Editor* and the *Executor*, respectively. Notably, when a medium-long context length LLM such as `WizardLM2` acts as the *Planner* and replaces the role of *Editor* or *Navigator*, we observe a more severe drop in the resolving rate. This is attributed to these roles requiring continuous interaction with the environment, necessitating a long context.

Model		SWE-bench Tiny	
		% Pass Rate	$ Avg. Cost
GPT-4o	HyperAgent	15.00	0.42
	w/o Navigator	7.00	2.81
	w/o Editor	11.00	1.92
	w/o Executor	14.00	0.75
WizardLM2	HyperAgent	13.00	0.31
	w/o Navigator	4.00	1.21
	w/o Editor	7.00	0.51
	w/o Executor	13.00	0.38

Table 8 | Ablation study on different agent role's contribution on SWE-bench Tiny

`go_to_definition`		`open_file`		`code_search`		`auto_repair_editor`	
Used	$9.00_{\downarrow 6.0}$	Used	$9.00_{\downarrow 6.0}$	Used	$8.00_{\downarrow 6.0}$	Used	$8.00_{\downarrow 7.0}$
w/ search	15.00	w/ annotated lines	$11.00_{\downarrow 4.0}$	w/ preview	$11.00_{\downarrow 3.0}$	w/ linting feedback	$11.00_{\downarrow 4.0}$
No usage	$12.0_{\downarrow 3.0}$	w/ keyword summary	15.00	w/ ranking	14.00	w/ repairing	15.00
		No usage	$4.0_{\downarrow 11.0}$	No usage	$3.0_{\downarrow 11.0}$	No usage	$1.0_{\downarrow 14.0}$

Table 9 | Ablation result on resolving performance on SWE-Bench Tiny with different key tool designs

7.2. Analysis of Tool Design

We investigated the improvements brought by our major design choices in the tool's interface and functionality. An ablation study was conducted on the functionalities of `go_to_definition`, `auto_repair_editor`, `open_file`, and `code_search` using SWE-bench Tiny. For each tool, we evaluated the overall performance when the tool is utilized versus when it is not, as shown in Table 9. A crucial finding for `go_to_definition` is that the LLM agent struggles to effectively use this IDE-like feature. It requires exact line and column numbers and the precise symbol name, which demands precise localization of character positions. Despite supporting annotated line numbers, the agent often fails and retries multiple times. However, incorporating a proximity-based search process, allowing the agent to approximate specifications, significantly improves performance (from 9% without search to 15% with search). For `open_file`, small LLMs like Claude Haiku tend to scroll up and down multiple times to find desired snippets by continuously increasing start_line and end_line, leading to out-of-context length issues. We addressed this by adding an additional input field keywords, allowing the LLM to search keywords inside the file. This enables the tool to quickly localize the positions of keywords inside the file and display the surrounding lines, increasing the resolving rate by 3%. Without `code_search`, the *Navigator* faces significant challenges in swiftly identifying necessary objects, resulting in a substantially lower performance rate of 3% compared to 8% when the tool is employed. Enhancing the output to include partial surrounding context around the keyword enables the *Navigator* to make more informed decisions, improving performance from 8% to 11%. Prioritizing search results for key objects such as functions and classes, and re-ranking these results further enhances overall performance, increasing it from 11% to 14%. We observed a substantial enhancement (8% to 11%) when providing Python linting feedback[4] to the *Editor* whenever it produces a patch via `auto_repair_tool`. By enabling an internal LLM of the tool to autonomously refine the generated patch and attempt to fix any encountered errors, the quality of the patch improves, leading to a 4% increase in resolving rate.

[4]We use flake8 for providing syntax errors.

7.3. Agent Behavior

We analyzed the frequency of each agent role requested by the *Planner* throughout the issue resolution process. Figure 3 illustrates a typical pattern where the *Planner* is most active at the beginning of the resolution process, gathering relevant information about the codebase environment. Subsequently, the *Editor* is frequently used to generate patches, often immediately following the *Navigator*, with notable peaks at Iterations 4 and 8. Finally, the *Executor* is requested more frequently in the later iterations to verify the results by executing tests. It is noteworthy that, in the first iteration, there is a small peak indicating that the *Executor* is requested to reproduce the issue.

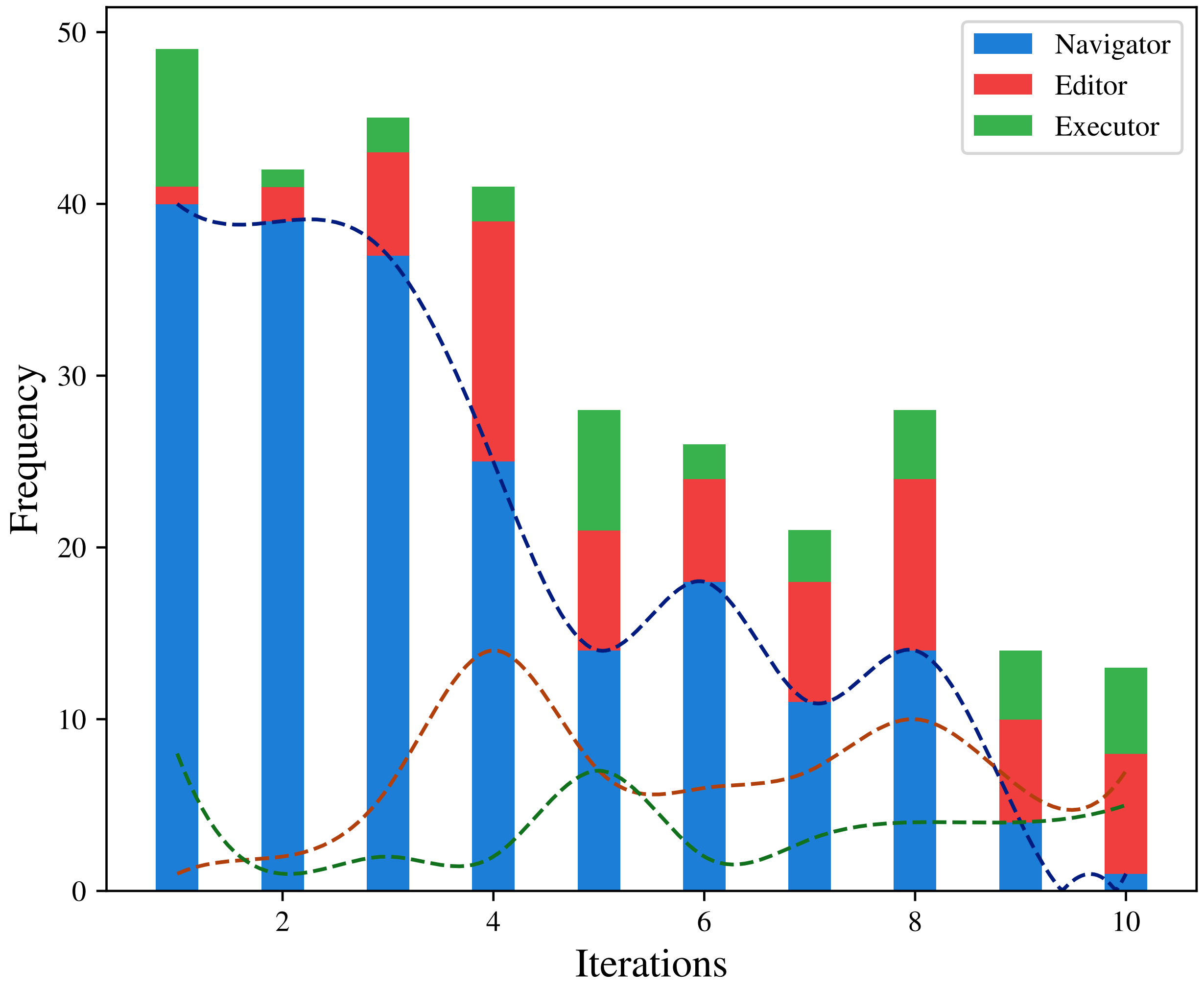


Figure 3 | Frequency of agent role requests by the *Planner* throughout the issue resolution process. The graph illustrates the typical pattern of agent activities: the *Planner* is most active at the beginning, gathering codebase information; the *Editor* is frequently used for patch generation, often following the *Navigator*, with peaks at Iterations 4 and 8; the *Executor* is more active in later iterations for result verification, with a small peak in the first iteration for issue reproduction.

8. Conclusion

In this paper, we introduced HyperAgent , a generalist multi-agent system designed to address a wide range of software engineering tasks. By closely mimicking typical software engineering workflows, HyperAgent incorporates stages for analysis, planning, feature localization, code editing, and execution/verification. Our extensive evaluations across diverse benchmarks, including GitHub issue resolution, code generation at repository-level scale, and fault localization and program repair, demonstrate that HyperAgent not only matches but often exceeds the performance of specialized systems.

The success of HyperAgent highlights the potential of generalist approaches in software engineering, offering a versatile tool that can adapt to various tasks with minimal configuration changes. Its design emphasizes generalizability, efficiency, and scalability, making it well-suited for real-world software development scenarios where tasks can vary significantly in complexity and scope.

Future work could explore the integration of HyperAgent with existing development environments and version control systems to further streamline the software engineering process. Additionally, investigating the potential of HyperAgent in more specialized domains, such as security-focused code review or performance optimization, could expand its applicability. Enhancing the system's explainability and providing more detailed insights into its decision-making process could also improve trust and adoption among developers. Finally, exploring techniques to continually update and refine the system's knowledge base with the latest programming paradigms and best practices could ensure its long-term relevance in the rapidly evolving field of software engineering.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*, 2023.

Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

Daman Arora, Atharv Sonwane, Nalin Wadhwa, Abhav Mehrotra, Saiteja Utpala, Ramakrishna Bairi, Aditya Kanade, and Nagarajan Natarajan. Masai: Modular architecture for software-engineering ai agents. *arXiv preprint arXiv:2406.11638*, 2024.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021.

Matej Balog, Alexander L Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. Deepcoder: Learning to write programs. *arXiv preprint arXiv:1611.01989*, 2016.

Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. *arXiv preprint arXiv:2403.17134*, 2024.

Nghi DQ Bui and Lingxiao Jiang. Hierarchical learning of cross-language mappings through distributed vector representations for code. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*, pages 33–36, 2018.

Nghi DQ Bui, Yijun Yu, and Lingxiao Jiang. Treecaps: Tree-based capsule networks for source code processing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 30–38, 2021.

Nghi DQ Bui, Yue Wang, and Steven Hoi. Detect-localize-repair: A unified framework for learning to debug with codet5. *arXiv preprint arXiv:2211.14875*, 2022.

Nghi DQ Bui, Hung Le, Yue Wang, Junnan Li, Akhilesh Deepak Gotmare, and Steven CH Hoi. Codetf: One-stop transformer library for state-of-the-art code llm. *arXiv preprint arXiv:2306.00029*, 2023.

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.

Bruno Castro, Alexandre Perez, and Rui Abreu. Pangolin: an sfl-based toolset for feature localization. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1130–1133. IEEE, 2019.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397*, 2022.

Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. Coder: Issue resolving with multi-agent and task graphs. *arXiv preprint arXiv:2406.01304*, 2024.

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*, 2021a.

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*, 2021b.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. *arXiv preprint arXiv:2304.05128*, 2023.

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*, 36, 2024.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

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*, 2024a.

Yu Gu, Yiheng Shu, Hao Yu, Xiao Liu, Yuxiao Dong, Jie Tang, Jayanth Srinivasa, Hugo Latapie, and Yu Su. Middleware for llms: Tools are instrumental for language agents in complex environments. *arXiv preprint arXiv:2402.14672*, 2024b.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. UniXcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio, editors, *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 7212–7225, Dublin, Ireland, May 2022a. Association for Computational Linguistics. doi: 10.18653/v1/2022.acl-long.499. URL https://aclanthology.org/2022.acl-long.499.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022b.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y Wu, YK Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

Nam Le Hai, Dung Manh Nguyen, and Nghi DQ Bui. Repoexec: Evaluate code generation with a repository-level executable benchmark. *arXiv preprint arXiv:2406.11927*, 2024.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, et al. Measuring coding challenge competence with apps. *arXiv preprint arXiv:2105.09938*, 2021.

Dávid Hidvégi, Khashayar Etemadi, Sofia Bobadilla, and Martin Monperrus. Cigar: Cost-efficient program repair with llms. *arXiv preprint arXiv:2402.06598*, 2024.

Sirui Hong, Xiawu Zheng, Jonathan Chen, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, et al. Metagpt: Meta programming for multi-agent collaborative framework. *arXiv preprint arXiv:2308.00352*, 2023.

Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

Dong Huang, Qingwen Bu, Jie M Zhang, Michael Luck, and Heming Cui. Agentcoder: Multi-agent-based code generation with iterative testing and optimisation. *arXiv preprint arXiv:2312.13010*, 2023.

Md Ashraful Islam, Mohammed Eunus Ali, and Md Rizwan Parvez. Mapcoder: Multi-agent code generation for competitive problem solving. *arXiv preprint arXiv:2405.11403*, 2024.

Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2023.

René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1(FSE):1424–1446, 2024.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Wen-tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation. In *International Conference on Machine Learning*, pages 18319–18345. PMLR, 2023.

Jia Li, Ge Li, Yunfei Zhao, Yongmin Li, Huanyu Liu, Hao Zhu, Lecheng Wang, Kaibo Liu, Zheng Fang, Lanshen Wang, Jiazheng Ding, Xuanming Zhang, Yuqi Zhu, Yihong Dong, Zhi Jin, Binhua Li, Fei Huang, and Yongbin Li. Deveval: A manually-annotated code generation benchmark aligned with real-world code repositories, 2024.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023.

Xia Li, Wei Li, Yuqun Zhang, and Lingming Zhang. Deepfl: Integrating multiple fault diagnosis dimensions for deep fault localization. In *Proceedings of the 28th ACM SIGSOFT international symposium on software testing and analysis*, pages 169–180, 2019.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and LINGMING ZHANG. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023a.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2023b.

Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Zengxian Yang, Kaikai An, et al. Ml-bench: Large language models leverage open-source libraries for machine learning tasks. *arXiv preprint arXiv:2311.09835*, 2023c.

Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 664–676, 2021.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, et al. Starcoder 2 and the stack v2: The next generation. *arXiv preprint arXiv:2402.19173*, 2024.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Jabier Martinez, Nicolas Ordoñez, Xhevahire Tërnava, Tewfik Ziadi, Jairo Aponte, Eduardo Figueiredo, and Marco Tulio Valente. Feature location benchmark with argouml spl. In *Proceedings of the 22nd International Systems and Software Product Line Conference-Volume 1*, pages 257–263, 2018.

Gabriela K Michelon, Bruno Sotto-Mayor, Jabier Martinez, Aitor Arrieta, Rui Abreu, and Wesley KG Assunção. Spectrum-based feature localization: a case study using argouml. In *Proceedings of the 25th ACM International Systems and Software Product Line Conference-Volume A*, pages 126–130, 2021.

Niklas Muennighoff, Qian Liu, Armel Randy Zebaze, Qinkai Zheng, Binyuan Hui, Terry Yue Zhuo, Swayam Singh, Xiangru Tang, Leandro Von Werra, and Shayne Longpre. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*, 2023.

King Han Naman Jain, 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.

Minh Huynh Nguyen, Thang Phan Chau, Phong X Nguyen, and Nghi DQ Bui. Agilecoder: Dynamic collaborative agents for software development based on agile methodology. *arXiv preprint arXiv:2406.11912*, 2024.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474*, 2022.

Nikhil Pinnaparaju, Reshinth Adithyan, Duy Phung, Jonathan Tow, James Baicoianu, Ashish Datta, Maksym Zhuravinskyi, Dakota Mahan, Marco Bellagente, Carlos Riquelme, et al. Stable code technical report. *arXiv preprint arXiv:2404.01226*, 2024.

Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. Communicative agents for software development. *arXiv preprint arXiv:2307.07924*, 2023.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, et al. Chatdev: Communicative agents for software development. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15174–15186, 2024.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Minghao Shao, Boyuan Chen, Sofija Jancheska, Brendan Dolan-Gavitt, Siddharth Garg, Ramesh Karri, and Muhammad Shafique. An empirical evaluation of llms for solving offensive security challenges. *arXiv preprint arXiv:2402.11814*, 2024.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Victor Sobreira, Thomas Durieux, Fernanda Madeiral, Martin Monperrus, and Marcelo de Almeida Maia. Dissection of a bug dataset: Anatomy of 395 patches from defects4j. In *2018 IEEE 25th international conference on software analysis, evolution and reengineering (SANER)*, pages 130–140. IEEE, 2018.

Amitayush Thakur, Yeming Wen, and Swarat Chaudhuri. A language-agent approach to formal theorem-proving. *arXiv preprint arXiv:2310.04353*, 2023.

Hung To, Minh Nguyen, and Nghi Bui. Functional overlap reranking for neural code generation. In *Findings of the Association for Computational Linguistics ACL 2024*, pages 3686–3704, 2024.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. *arXiv preprint arXiv:2402.01030*, 2024a.

Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F Xu, and Graham Neubig. Mconala: a benchmark for code generation from multiple natural languages. *arXiv preprint arXiv:2203.08388*, 2022.

Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks. *arXiv preprint arXiv:2401.12869*, 2024b.

Ratnadira Widyasari, Sheng Qin Sim, Camellia Lok, Haodi Qi, Jack Phan, Qijin Tay, Constance Tan, Fiona Wee, Jodie Ethelda Tan, Yuheng Yieh, et al. Bugsinpy: a database of existing bugs in python programs to enable controlled testing and debugging studies. In *Proceedings of the 28th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 1556–1560, 2020.

W Eric Wong, Vidroha Debroy, Yihao Li, and Ruizhi Gao. Software fault localization using dstar (d*). In *2012 IEEE Sixth International Conference on Software Security and Reliability*, pages 21–30. IEEE, 2012.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents. *arXiv preprint arXiv:2407.01489*, 2024.

Frank F Xu, Uri Alon, Graham Neubig, and Vincent Josua Hellendoorn. A systematic evaluation of large language models of code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, pages 1–10, 2022.

John Yang, Akshara Prabhakar, Shunyu Yao, Kexin Pei, and Karthik R Narasimhan. Language agents as hackers: Evaluating cybersecurity skills with capture the flag. In *Multi-Agent Security Workshop@ NeurIPS'23*, 2023.

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*, 2024a.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. *Advances in Neural Information Processing Systems*, 36, 2024b.

He Ye and Martin Monperrus. Iter: Iterative neural repair for multi-location patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. Selfapr: Self-supervised program repair with test execution diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yuchi Ma, Guangtai Liang, Ying Li, Qianxiang Wang, and Tao Xie. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–12, 2024.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions. *Advances in Neural Information Processing Systems*, 36:31466–31523, 2023a.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation. In *OPT 2023: Optimization for Machine Learning*, 2023b.

Fengji Zhang, Bei Chen, Yue Zhang, Jacky Keung, Jin Liu, Daoguang Zan, Yi Mao, Jian-Guang Lou, and Weizhu Chen. Repocoder: Repository-level code completion through iterative retrieval and generation, 2023.

Kexun Zhang, Weiran Yao, Zuxin Liu, Yihao Feng, Zhiwei Liu, Rithesh Murthy, Tian Lan, Lei Li, Renze Lou, Jiacheng Xu, et al. Diversity empowers intelligence: Integrating expertise of software engineering agents. *arXiv preprint arXiv:2408.07060*, 2024a.

Shaokun Zhang, Jieyu Zhang, Jiale Liu, Linxin Song, Chi Wang, Ranjay Krishna, and Qingyun Wu. Training language model agents without modifying language models. *arXiv preprint arXiv:2402.11359*, 2024b.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024c.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. Opencodeinterpreter: Integrating code generation with execution and refinement. *arXiv preprint arXiv:2402.14658*, 2024.

Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. An empirical study of fault localization families and their combinations. *IEEE Transactions on Software Engineering*, 47 (2):332–347, 2019.