# Unified Software Engineering Agent as AI Software Engineer

Leonhard Applis*
l.applis@nus.edu.sg
National University of Singapore
Singapore

Yuntong Zhang*†
yuntong@comp.nus.edu.sg
National University of Singapore
Singapore

Shanchao Liang
liang422@purdue.edu
Purdue University
West Lafayette, Indiana, USA

Nan Jiang
jiang719@purdue.edu
Purdue University
West Lafayette, Indiana, USA

Lin Tan
lintan@purdue.edu
Purdue University
West Lafayette, Indiana, USA

Abhik Roychoudhury
abhik@comp.nus.edu.sg
National University of Singapore
Singapore

## ABSTRACT

The growth of Large Language Model (LLM) technology has raised expectations for automated coding. However, software engineering is more than coding and is concerned with activities including maintenance and evolution of a project. In this context, the concept of LLM *agents* has gained traction, which utilize LLMs as reasoning engines to invoke external tools autonomously. But is an LLM agent the same as an AI software engineer? In this paper, we seek to understand this question by developing a Unified Software Engineering agent or USEagent. Unlike existing work which builds specialized agents for specific software tasks such as testing, debugging, and repair, our goal is to build a unified agent which can orchestrate and handle multiple capabilities. This gives the agent the promise of handling complex scenarios in software development such as fixing an incomplete patch, adding new features, or taking over code written by others. We envision USEagent as the first draft of a future AI Software Engineer which can be a team member in future software development teams involving both AI and humans. To evaluate the efficacy of USEagent, we build a Unified Software Engineering bench (USEbench) comprising of myriad tasks such as coding, testing, and patching. USEbench is a judicious mixture of tasks from existing benchmarks such as SWE-bench, SWT-bench, and REPOCOD. In an evaluation on USEbench consisting of 1,271 repository-level software engineering tasks, USEagent shows improved efficacy compared to existing general agents such as OpenHands CodeActAgent. For specific tasks such as issue resolution in SWE-bench, USEagent demonstrates efficacy comparable to agents such as AutoCodeRover (which are focused on software maintenance), while maintaining applicability to a broader range of software engineering tasks. There exist gaps in the capabilities of USEagent for certain coding tasks, which provides hints on further developing the AI Software Engineer of the future.

---

*Joint first authors, ordered alphabetically.
†Corresponding author, queries about the paper can be sent to Yuntong Zhang.

---

## 1 INTRODUCTION

Large language models (LLMs) have shown promise in coding, reasoning, and problem solving. The model capability itself has improved significantly in terms of reasoning tasks, by training the models to always use reasoning paradigms such as chain of thoughts, without needing further prompting, as shown by GPT o1 [25] and its descendant models such as o3. On top of improved foundation models, the software engineering community is distilling known best practices for downstream tasks into amplifiers for LLMs through agentic systems [39]. An LLM agent for software engineering invokes various interfaces including testing and analysis tools autonomously driven by an LLM reasoning agent.

While agentic systems inspire researchers [5, 39], start-ups [3, 26] and major companies [28, 33], we currently observe a strong specialization in the emerging technologies among the LLM agents for software. Agentless [37] performs program repair, Large Language Monkeys [9] produce unit tests, ExecutionAgent [8] performs a project setup, and so forth. These specialized agents are reasonable efforts, especially as they require specialized metrics and datasets. However, with many agents comes the need to manage and maintain them as part of the future development environments!

We propose a unified Software Engineering agent (USEagent) representing a consolidated agentic capability for software engineering tasks. Individual SE tasks could be coordinated to result in a more effective ensemble of tasks: better test-generation should benefit reproduction in program repair, review of patches should lead to better generated code, etc. With an integrated software engineering agent, it becomes much more feasible to co-opt it in future development environments [2].

As a foundation, we need a unified dataset of challenging SE tasks to test our unified agentic capabilities, for which we build Unified Software Engineering Bench, or USEbench for short. Recently, the SWE-bench dataset [18] has been proposed that captures a set of GitHub issues depicted in natural language and requires bug fixes or feature additions in software projects. Thus the SWE-bench dataset is replete with challenges in software maintenance for various real-life software projects. Our proposed benchmark USEbench is a meta-benchmark composing a diverse set of software engineering tasks (such as code generation, program repair, test generation) behind a unified application programming interface (API). Building a unified API is key, since it allows us to start thinking of a unified agentic capability which can handle different software engineering tasks. After constructing USEbench, we expand two popular agentic systems, one from industry and one from academia, AutoCodeRover [42] and OpenHands CodeActAgent [34, 35] to solve the tasks in USEbench. We consider this a natural evolution or progression of the flurry of research proposing manifold different agents for different software engineering tasks. Existing specialized agents usually employ a fixed workflow and approach the given task in a few pre-defined steps. For example, AutoCodeRover tackles program repair tasks using a fixed two-phase workflow consisting of fault localization and patch generation. Now, what does it take to generalize an existing program repair agent, with fixed actions and pre-defined workflow, so that it can act as a unified software engineering agent (USEagent)? For each task or problem in USEbench, the agentic systems are only provided with a description (i.e., a bug report or the documentation of a method to generate) as well as access to a containerized environment of the project. As such, our meta-benchmark reveals inherent challenges regarding *task-identification*, *workflow-configuration* and *measuring progress* (esp. determining end-criteria).

To address these requirements, we equip AutoCodeRover with a `Meta-Agent`, which is instructed to orchestrate the appropriate agents and construct a workflow *on the fly*. Over the course of this workflow, the `Meta-Agent` utilizes available actions to construct and maintain a project-state, constituting a structured consensus memory over the trajectories LLM components. Since OpenHands CodeActAgent is a general-purpose agent designed to solve a variety of tasks, we consider it for baseline comparison, with no architectural changes applied.

We report the efficacy of both agentic systems on USEbench including PASS@1 and PASS@5 results. We perform a detailed error analysis as well as a manual inspection of *positives*, to identify false-positives and cases of over-fitting or memorization. On USEbench consisting of 1271 tasks including program repair, regression testing, code generation, and test generation, USEagent achieves an efficacy of 33.3%, which is higher than the 26.8% efficacy from the state-of-the-art general agent OpenHands CodeActAgent. Specifically, on software maintenance tasks in SWE-bench-verified, USEagent has an efficacy of 45.6% which is similar to the 46.2% efficacy of the specialized AutoCodeRover agent, while being applicable to more types of tasks. On test generation tasks that AutoCodeRover could not be applied to, USEagent achieves 31.8% efficacy, demonstrating its versatility. We also make the USEbench benchmark public, to encourage further research in design of AI software engineers.

## 2 UNIFIED SOFTWARE ENGINEERING BENCHMARK

To make progress towards building a unified agent for software engineering (USEagent), we first build a dataset of automated software engineering tasks. There is a rich environment of benchmarks, yet they are somewhat fragmented - they focus on specific types of software engineering tasks. Popular benchmarks encompass natural language issues for software (SWE-bench [18]), automated program repair (Defects4J [19]), code generation (REPOCOD [20]), test generation (SWT [23]) or documentation generation (CodeNet [27]), each accompanied by a metric to identify correct or optimal solutions. Next to their task, benchmarks often vary in scope too: the *programming* benchmarks (such as HumanEval [12] , CodeNet [27], BigCode [4], EvalPlus [21]) cover method-snippets or class-level code that is evaluated against known examples. The recently proposed SWE-bench [18] offers a new angle on repository-scale changes by introducing natural language issues for program repair instead of a known test-suite. However, due to the natural language, some issues were too ambigious or un-solvable — spurring efforts of OpenAI to create a human-verified subset, called SWE-bench-verified [24]. Solving such natural language issues automatically brings the vision of a future AI software engineer closer to the capabilities of a human software engineer.

In this paper, we propose USEbench, a unified software engineering benchmark as a meta-benchmark which moves beyond project-level code editing tasks suggested by SWE-bench. Our goal is to unify existing individual benchmark sets and design a comprehensive unified dataset capturing multiple SE tasks. Our proposal combines a set of existing benchmark sets, namely SWE-bench-verified [24], SWT [23], REPOCOD [20] and REPOTEST. Out of these SWE-bench-verified captures tasks which involve code editing or program modifications to achieve bug fixing / feature addition. SWT bench presents validation tasks for testing real-world code fixes in software projects. REPOCOD is a recent benchmark which captures code generation tasks in a software project. REPOTEST is a not previously published derivative of REPOCOD: While REPOCOD encompasses repository level code-generation after removing a method body; REPOTEST removes tests for a specified method-body and evaluation is based on code-coverage.

The exact composition of USEbench from the constituent benchmarks is shown in Table 1. Six different task types are covered, and for each task type several instances are included in USEbench. As a notable technical contribution, we provide a unified interface for agentic interaction with the unified benchmark. Previous benchmarks like SWE-bench provide a suitable interface for evaluation of code editing tasks *i.e.*, after providing a `.json` file tasks are streamlined; still there is significant work involved in interacting with a software project, or extracting data from the constituent files of a software project. Thus, LLM agents proposed to work on SWE-bench tasks need to make these adjustments to work with SWE-bench, e.g. see [26]. We would want our unified benchmark USEbench to possess a greater degree of usability, so that LLM agents can be evaluated and compared in terms of their efficacy in conducting the software engineering tasks in the benchmark. To alleviate the burden on researchers, the constituent benchmarks are conveniently plugged behind a common interface revolving

**Table 1: Summary of primitive and compound tasks unified in USEbench.**

| Benchmark | Task | Concept | Solving Criteria | # |
|---|---|---|---|---|
| Primitive Tasks | | | | |
| SWE-bench-verified | Program Repair | Adjust a program according to an issue. | Test-Suite Pass | 500 |
| SWT-bench-Lite | Regression Testing | Produce a test that asserts a required change, as presented in an issue | Patch-Coverage | 298 |
| REPOCOD-Lite | Code Generation | Generate a method body, matching the functionality outlined in documentation | Test-Suite Pass | 200 |
| REPOTEST-Lite | Test Generation | Test a given method to reach 100% coverage | Code-Coverage | 173 |
| Compound Tasks | | | | |
| Partial fix | Test Generation Program Repair | Enrich the SWE-Task with a previously failed, but promising patch | Test-Suite Pass | 100 |
| Feature Development | Code- & Test-Generation | Add both tests and code, evaluate against the newly produced code coverage | Code-Coverage | 46 |

around docker images, providing options to read files and execute commands, regardless of the task. Due to the unified nature of our benchmark USEbench, we can also imagine unified software engineering agent (USEagent) working on the tasks in USEbench. Such a unified agent will be able to achieve complex software engineering tasks, which are solved via a sequence of the software engineering task types covered in the benchmark USEbench.

We now show two concrete scenarios of more complex software engineering tasks constructed from primitive task types in USEbench, namely *incomplete fix* and *feature development*, elaborated in Table 1. To concretely see how the different task types in USEbench are combined into compound tasks, let us consider the scenario of handling an incomplete code fix. In such a scenario, we (i) test the code and find failing tests, (ii) generate a fix for the failed tests, (iii) test the fixed code and find more failed tests, and (iv) finally generate a fixed code which passes all tests. These four steps could be four different runs of a software engineering agent, and could combine different task types from USEbench, namely code fixing and test generation. The unified API of tasks in USEbench allows us to combine and alter tasks to capture more complex software engineering challenges, like incomplete fixes. Another scenario of a complex software engineering task is the complete addition of a feature to a codebase. This includes a problem description, and requires a functional method-body as well as accommodating tests. How to solve the task, as well as any form of decomposition, re-iteration or co-evolution is left up to the users of the benchmark. The final evaluation of this derived scenario will be against a hidden test-suite, as well as the generated tests.

## 3 BACKGROUND: AGENTS FOR SOFTWARE ENGINEERING

Agents typically utilize LLMs for decision making and content generation, along with autonomously invoked *tools* to interact with external entities. These tools allow the agent access to knowledge beyond the LLMs' training data, and allow the agent to influence the external environment through the tool interface. Agents often employ reasoning frameworks to orchestrate the LLM and the tool

usage. One of the popular reasoning frameworks used in agents is ReAct [39], in which the LLMs are instructed to reason about output-traces and invoke tools in an interleaved manner.

In software engineering, the target system that an agent interacts with is usually a software project. Another component of the external system is an execution environment, where software can be executed and tested. Existing software engineering agents (SE agents) proposed different ways to interact with the software project and the execution environment. AutoCodeRover [42] uses a set of project structure-aware tools (e.g. search_method_in_-class to navigate the software codebase and gather necessary code context. It further performs test execution and spectrum-based localization (SBFL) [36] to pinpoint more relevant code locations. SWE-Agent [38] employs file-based tools to navigate files in the project, for example find_file, open, scroll_down, and together they form an agent-computer interface [38]. OpenHands CodeActAgent [34, 35] provides a set of basic tools such as CmdRunAction to execute arbitrary bash commands inside a sandbox environment. A similar approach was followed by Google Jules agent [14]. Roughly summarized, the existing agents differ (intentionally) in how much structure and domain knowledge they introduce.

Another aspect of agent design is the decision framework. Existing SE agents have explored options for more controlled decision-making beyond the ReAct framework, in the context of software engineering. AutoCodeRover [42] and Agentless [37] targets software maintenance tasks, and they break the agent's execution into phases like context retrieval/localization and repair. Another example is RepairAgent [7], which restricts available tools through a finite-state machine. The different designs in existing SE agents raise an important question – how much *autonomy* should be given to the agent during decision making? On one hand, agents can follow a fine-tuned workflow for the targeted software engineering (SE) task (e.g. in AutoCodeRover, Agentless, RepairAgent). In this design, agents operate in fixed phases (e.g. localization is the first phase for program repair), which makes them more cost-effective [37, 42]. With the task broken down into phases, it also becomes easier to leverage program analysis techniques in each of the phases. However, agents with this design have less autonomy, and as a result,

they could not be directly applied to another kind of SE task. Applying a program repair agent to test generation would require drastic changes to its workflow. On the other hand, agents can be given more autonomy by having access to a general set of tools and the freedom to decide when to use these tools (e.g. in SWE-Agent and OpenHands CodeActAgent). This general design makes the agents applicable to many kinds of software engineering tasks. However, this generality makes it harder to exploit domain knowledge and program analysis in the agents. Moreover, when the task becomes complex and the agent execution gets longer, it is challenging to interpret the agent trajectory consisting of a list of general tool usages. In comparison, agents operating in "phases" are more interpretable, since we can inspect the results of each phase (e.g. what is the identified faulty location at the end of a localization phase).

In this work, our goal is to have an agent applicable to multiple types of software engineering task, while still utilizing domain-specific optimizations and interpretability. We name this agent Unified Software Engineering Agent (USEagent).

## 4 UNIFIED SOFTWARE ENGINEERING AGENT

The first step towards a unified software engineering agent is breaking down the fixed workflow of existing agents into components (i.e., phases) and add orchestration to decide which component to invoke based on the task type and *task state*. To achieve this, we propose a `Meta-Agent`, a central LLM-reasoning agent that orchestrates various components, henceforth called *actions*. The `Meta-Agent` and the actions together form a unified software engineering agent (USEagent) that is capable of solving several types of software engineering tasks with a higher level of autonomy. We identify several challenges in designing such a unified agent to solve tasks in USEbench.

The first challenge is to adapt to tasks beyond a fixed workflow. Previous agents extract actions such as context retrieval or patch generation as clear "units of work" from developer workflows and organize them into state-machines or fixed workflows. This rigid organization of actions restricts their adaptability to other tasks. To achieve high agent autonomy we allow the actions to be freely composed and introduce a `Meta-Agent` that orchestrates the actions using a ReAct-style loop, forming a *meta-layer* over the actions, shown in Figure 1. Starting from natural-language descriptions, the `Meta-Agent` invokes actions, observes the changes made by the action, reasons about the output, and decides on the next action to take. This framework enables *on-the-fly* composition of workflows, adjusting to different tasks, errors and changing states. ReAct-style decision making has been widely used in previous agents to orchestrate *tools* that directly interact with the codebase; which we extend towards *action* orchestration. The overall trajectory and choices of the `Meta-Agent` will construct an *action-graph*. In our action-graph, any node can be the start or end-point - for example, depending on the task, an action of "executing tests" can be the starting point or the last assurance before termination. The agent execution can be seen as an implicit construction of this action-graph on the fly.

The next challenge is to provide the `Meta-Agent` with a set of reliable, yet flexible actions that are suitable for composition. Existing general agents such as OpenHands CodeActAgent [35] and Google Jules [14] provide the LLM with a console to execute *any* command.
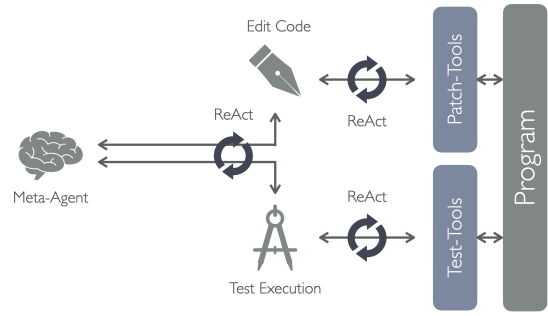


**Figure 1: Concept: Meta-Agent abstracting over actions.**

The action of executing a console command is extremely flexible, but hard to control. The agent's execution consists of a sequence of console commands, which are difficult for human developers to interpret and raise security concerns. Thus, we propose to design our actions at a coarser granularity, where each action encapsulates a "unit of work" that developers carry out during the software engineering process. Each action receives instructions from the `Meta-Agent` on *what outcomes it is supposed to achieve*, rather than *how it achieves them*. The `Meta-Agent` and individual actions communicate through this intent-based interface. An example action is `EditCode`, where the `Meta-Agent` specifies what should be edited in the code, but not the exact details of making the edits. This design results in a set of modularized actions with clear responsibilities, that can be tested, extended and maintained individually. Moreover, when one action is improved, the improvements propagate across all involved task types. We present a sample list of actions for unified software engineering agents in Section 4.1.

The third challenge is knowledge management: For complex tasks, the information provided *a priori* is not sufficient, and a central effort in solving tasks is identifying and producing relevant information. Once produced, this context should be made available to the agentic system, i.e. `Meta-Agent` and other actions. Existing work has defined this problem as *memory management* of agents [17, 43], where memory refers to historical information relevant to the current task. In literature on LLM agents, memory is often categorized into short-term (e.g., information within an ongoing conversation), long-term (e.g., historical data from previous sessions), and consensus memory (e.g., shared knowledge between multiple agents) [17]. In this work, we explicitly identify what information to memorize when designing a unified software engineering agent, and classify them into the different types of memory. We categorize the results of actions as short-term memory, which is only presented to the `Meta-Agent` to decide the next action. Project-specific knowledge, such as developer-written documentation, is considered long-term memory. These documents are embedded and stored in a vector database accessible for Retrieval-Augmented Generation (RAG) by the agent. Finally, artifacts generated by actions that are useful in guiding other actions constitute consensus memory. Such artifacts include program locations or code edits that can be notable to other actions. We introduce a structured *task state* that represents the consensus memory, which can be read and modified by different actions.
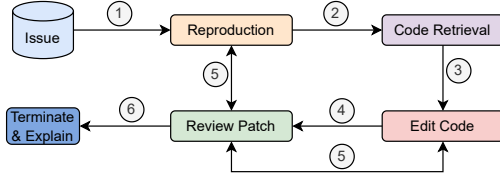
**Figure 2: Overview of AutoCodeRover. AutoCodeRover composes a fixed workflow for program maintenance tasks.**

## 4.1 Instantiating AutoCodeRover as USEagent

In this section, we highlight steps to build the unified SE agent USEagent that handles multiple types of SE tasks. We design our USEagent drawing inspiration from the open-source agent AutoCodeRover [29, 42], since AutoCodeRover operates in different phases which are suitable as starting points for actions. We elaborate the design of USEagent and flesh out our proposed solutions to the challenges in Section 4.

AutoCodeRover is an LLM agent that targets program maintenance tasks. Figure 2 shows the overall workflow of AutoCodeRover. Starting from a natural-language description of a software issue, AutoCodeRover first writes an executable self-contained test that reproduces the issue (Step ①). The stacktrace of the reproducer test, together with the task description, is sent to a context retrieval component to identify relevant code locations (Step ②). The relevant program locations are made available to the *Edit Code* component for writing a candidate patch (Step ④). With both LLM-generated patch and reproducer test, the *Review Patch* component then executes the tests on the patched program, decides whether the are correctly written, and iteratively improves them (Step ⑤). Finally, the AutoCodeRover workflow finishes when the reviewer approves a patch or the execution reaches a pre-defined round limit (Step ⑥). AutoCodeRover proved effective for resolving software issues such as bug fixing [29, 42]. However, these fixed transitions among components make agents like AutoCodeRover not generalizable to task types that require a different workflow. For example, in a task like test-suite generation, steps such as Reproduction and Review Patch are irrelevant, requiring the design of a new workflow or even a new agent.

Instead of designing a new agent for each task type, our goal is to build *one* general agent that handles multiple task types. To achieve this, we disassemble the workflow of AutoCodeRover and reassemble it into USEagent shown in Figure 3. We adapt components in AutoCodeRover such as Code Retrieval into *actions* that can be freely composed by a Meta-Agent for different task types. Given the task description that mentions the task type in natural language, the Meta-Agent uses ReAct-style reasoning to select next actions to execute. Short-term feedback such as action execution results are reflected directly to the Meta-Agent, while artifacts that should be preserved longer are stored into a *task state* representing the consensus memory among actions. We next discuss each component of USEagent in more details.

*Actions.* Given the tasks of program repair, regression testing, code generation, and test generation in USEbench, we identify a
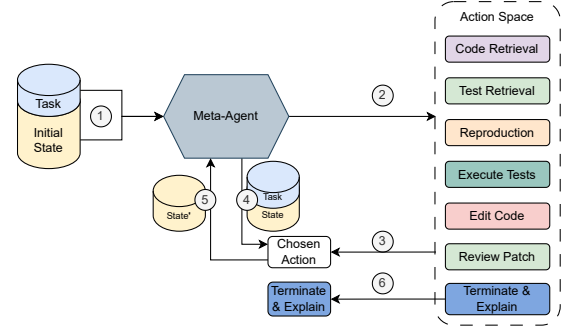


**Figure 3: Overview of the USEagent and workflow. The Meta Agent chooses available actions, provides the state and retrieves a altered state until termination is chosen.**

priming set of actions for USEagent. The set of actions are shown in Table 2. Each action provides the Meta-Agent with an interface that specifies its description, input, and output, but abstracts away the underlying execution details. In addition to interacting with the Meta-Agent through input and output, each action can also read from and write to the task state.

Many of the actions in Table 2 are conceptually inspired by those used in existing agents such as AutoCodeRover (Figure 2). On top of the existing agents, we designed a few new actions that are essential for the task types in USEbench. These new actions include TestRetrieval and ExecuteTests. The TestRetrieval action explores the codebase and retrieves testcases relevant to the current task. It works similarly compared to the existing CodeRetrieval, which employs a set of search tools such as search_func to search for relevant code units in the program. The ExecuteTests action executes parts of the project test suite, which is useful for validating code/test changes by execution. Current agents focusing on SWE-bench usually assume the commands to run existing unit tests are given as an input to the agent [29, 37]; however, this assumption requires additional manual configuration when setting up the agents on new projects. Our ExecuteTests action does not assume the test commands to be given, and instead it queries the project documentation files to retrieve the project-specific commands through RAG. In addition, we added a special Finish action that signals the termination of the agent execution. The Finish action is invoked with an argument specifying the final result for the current task. Unlike other actions which encapsulates concrete workflows, the Finish action only terminates the agent and outputs its argument as the final result from the agent.

These actions can be invoked multiple times and combined in various ways to complete a given task. For example, for a code generation task, the Meta-Agent may first invoke CodeRetrieval to understand the surrounding code context, invoke EditCode to draft the implementation, invoke TestRetrieval to find out where existing tests for this code component resides, and then invoke EditCode to write new tests for the newly added code. It may then enter a refinement loop of improving the new code and tests using the ExecuteTests and EditCode actions. We note that this sequence

**Table 2: Details of actions used in USEagent. `Diff` denotes a Patch to the project, and subscript specifies modifiers. E.g. $\text{Diff}_{\text{Code}}$ expresses a change to program code, instead of test code.**

| Action | Description | Action Signature | Interaction with Task State |
|---|---|---|---|
| CodeRetrieval | Collect program code relevant to the task description. | $\text{Description} \rightarrow \text{Locations}_{\text{Code}}$ | Write: $\text{Locations}_{\text{Code}}$ |
| TestRetrieval | Collect test code relevant to the task description. | $\text{Description} \rightarrow \text{Locations}_{\text{Tests}}$ | Read: $\text{Locations}_{\text{Code}}$<br>Write: $\text{Locations}_{\text{Test}}$ |
| Reproduction | Generate a system-level standalone reproduction test. | $\text{Description} \rightarrow \text{Diff}_{\text{Test}}$ | Write: $\text{Test}_{\text{Reprod.}}$<br>Write: $\text{Output}(\text{Test}_{\text{Reprod.}})$ |
| ExecuteTests | Execute a subset of the program test suite | $\text{List[TestFile]} \rightarrow \text{Summary(Results)}$ | Read: $\text{Locations}_{\text{Test}}$<br>Write: $\text{Command}_{\text{Test}}$ |
| EditCode | Make modifications across the codebase, including test-code. | $\Delta\text{Behavior} \times \text{Location} \rightarrow \text{Diff}$ | Read: $\text{Locations}_{\text{Code}}$<br>Read: $\text{Locations}_{\text{Test}}$<br>Write: $\text{Diff}$ |
| ReviewPatch | Review and iteratively improve a generated patch and generated test. | $\text{Test}_{\text{Reprod.}} \times \text{Diff}_{\text{Code}} \rightarrow \text{Bool}$ | Read: $\text{Diff}_{\text{Code}}$<br>Read: $\text{Test}_{\text{Reprod.}}$<br>Write: $\text{Diff}_{\text{Code}}$ |
| Terminate | Finish the agent execution and select a final solution. | $\text{State} \rightarrow \text{Diff}$ | - |

of action invocation can happen without prior configuration, as we will show in section 6.3.

*Task State.* Our next contribution towards a USEagent is the design of a *task state*. The task state represents the *consensus* memory among the actions - each action can write new artifacts generated by them into the task state so that the artifacts can be visible to other actions. Our task state captures essential artifacts involved in a software debugging process, including relevant locations, test execution information, and prior attempts of code modifications. Specifically, the task state $S$ is defined as: $S = (L_c, L_t, R_{exec}, DS)$. Here, $L_c$ represents the relevant code locations (e.g., program methods and classes); $L_t$ represents the test locations (e.g., unit test methods). The locations are identified by the retrieval actions, and are utilized by downstream code editing actions. $R_{exec}$ contains the execution results of different tests against different candidate patches. These execution information provides guidance to other actions in improving partial solutions and selecting the final solution. The last part of the task state is a *diff store* ($DS$), which records all the generated diff contents from the EditCode, Reproduction and ReviewPatch actions. These diff contents include modifications to both the program code and the test code. The diff store is necessary to allow for versatile invocations of the ExecuteTests and EditCode actions. By selecting different subsets of diffs in $DS$ and passing them to ExecuteTests, tests can be executed on different versions of the project after applying the selected diffs. Similarly, EditCode can first apply an existing patch from $DS$, and subsequently introduce additional modifications, thereby enhancing partial patches. Concretely, when an action is implemented, the implementation defines which part of the action output will be stored into the task state.

*MetaAgent.* The last important addition is the Meta-Agent which orchestrates the various actions. As shown in Figure 3, Meta-Agent takes in the task description and an initial empty task state (Step ①), and then iteratively selects an action from the action space to

execute (Step ②). Each action represents an encapsulated workflow to complete a "unit of work", and exposes its input/output interface to the Meta-Agent. At each step of action selection, we present the Meta-Agent with the current task state, the overall task description, as well as the output from the previously invoked action. The invoked action modifies the task state and retunrs it back to the Meta-Agent (Step ⑤). This interleaving of reasoning (i.e. select the next action based on current output and state) and action (i.e. execute the actual action to obtain new observations) forms a ReAct-style loop at the action level. When the Meta-Agent deems that one of the diffs in the diff store is a satisfactory solution to the given task, it invokes the Finish action to end the agent execution and output the final solution. In the case where the Meta-Agent could not decide on the final solution (i.e. could not invoke the Finish action) after a pre-defined limit for the ReAct-loop, we ends the execution and invoke the LLM to select one of the candidates in the diff store as the final solution.

## 4.2 Configuring OpenHands

OpenHands CodeActAgent [35] is a general-purpose agent designed to solve general tasks[1]. It utilizes a structured controller-agent-runtime architecture, where the AgentController acts as a supervisor, enforcing operational constraints (such as conversation iterations and budget) and managing the agent's lifecycle (start, stop, pause). The CodeActAgent makes decisions, interprets LLM responses, and converts them into actions to interact with the runtime, which is an isolated sandbox environment. Since CodeActAgent does not rely on a predefined task-specific workflow, it can be applied to a wide range of software engineering tasks, including those in USEbench, without requiring modifications to its architecture.

During execution, OpenHands CodeActAgent operates iteratively, generating actions, executing them, and processing observations before determining the next step. The actions include:

---

[1]Note: OpenHands is also developing CodeActSWEAgent, specialised on SWE-Bench. We purposefully chose the non-specialised implementation for USEbench. The entries on the SWE-Bench leaderboards are to our knowledge results from CodeActSWEAgent.

- `CmdRunAction`: execute Linux bash commands.
- `IPythonRunCellAction`: execute Python code in Jupyter or IPython environment.
- `FileEditAction`: read, write, or edit files in the runtime.
- `AgentFinishAction`: signal task completion or termination.

The execution of `CodeActAgent` terminates when the agent issues an AgentFinishAction, reaches resource limits (e.g., maximum iterations), or encounters an error. In practice, we design different prompt templates for each task type as input to OpenHands `CodeActAgent`, incorporating both the task type and task description. Compared to `USEagent`, the OpenHands `CodeActAgent` employs a similar reasoning framework, but the actions operate on a lower-level such as execution of a single bash command. It follows a single ReAct-loop centered around a more open set of available actions. This free-wheeling approach is an alternative to the more structured and pronounced options in Section 4.1. These two designs demonstrate a trade-off: allowing arbitrary bash commands makes the agent more flexible, while using structured actions improves interpretability and control over the agent.

## 5 RESEARCH QUESTIONS & EXPERIMENT SETUP

Our first research question focuses on evaluating and comparing agentic systems for tasks beyond program repair.

> **RQ1: Efficacy of Agentic Systems**
> How well do state-of-the-art agentic systems perform across the diverse software engineering tasks in USEbench?

We address **RQ1** by applying USEagent and OpenHands to all data points outlined in Section 2 and report a PASS@1. In addition to the efficacy, we also report an investigation of the major obstacles for the individual benchmarks. To understand the effect of randomness, we report PASS@5 on a significant subset of the data points. We also examine the feasibility of using open-source models as the backend of `USEagent`, by evaluating `USEagent` with DeepSeek-V3.

A known issue and motivation for our next research question is overfitting in plausible patches. Such patches pass the evaluation-criteria by bypassing the harness with *tricks* rather than benign functionality. Sometimes, actual solutions are checks for edge-cases - these can only be distinguished from overfitting by consulting the ground truth (i.e., the gold patch). We aim to identify and quantify overfitting solutions.

> **RQ2: Analysis of Plausible Patches**
> What is the rate of overfitting among the *resolved* patches?

We sample a significant subset[2] of solved data points and manually examine whether the plausible solution is overfitting. During this manual investigation, we also look for *memorization* in plausible patches, i.e. patches that perfectly mirror the ground truth developer patch or contain suspicious content.

As described in Section 4, one milestone is to make the agent adapt autonomously to different tasks. We study this adaptation by investigating the resulting sequence of actions in RQ3:

> **RQ3: Effectiveness of self-configuration and actions**
> Can USEagent self-configure based on the task type while maintaining reasonable efficacy? How does the addition of new actions affect the efficacy?

We present the sequence of actions of the `USEagent` and accumulate their patterns per task. We showcase examples on how these top-level patterns translate into commands, queries and edits. Furthermore, to quantitatively understand the effect of using dynamic decision making, we perform an ablation study by using a static action sequence for each task type. We also perform an ablation study by removing one of the newly added actions `ExecuteTests`.

Lastly, we identify open challenges in the current unified software engineering agent. We formulate key difficulties and discuss potential future directions to address them.

> **RQ4: Open challenges for agentic systems**
> What are the challenges faced by the current unified agent in tasks in USEagent? Can we identify actions to remediate them?

*Experiment setup.* We evaluate agentic systems such as `USEagent` and OpenHands `CodeActAgent` on the full `USEbench` which consists of 1271 datapoints. In most of our experiments, agentic systems use Anthropic Claude 3.5 Sonnet v2 (claude-3-5-sonnet-20241022) as the backend LLM. Additional experiments with DeepSeek-V3 invokes the DeepSeek-V3 model through the Openrouter API[3]. For `USEagent`, we set the maximum rounds of action invocation from the `Meta-Agent` to 20, and force the agent to end execution and select a candidate solution. We set the model temperature to zero in `USEagent`. To better understand the effect of randomness, we randomly sampled a statistically significant subset (295 instances) to report PASS@5 results from both agentic systems.

## 6 RESULTS

### 6.1 RQ1: Efficacy of Agentic Systems

An overview of the efficacy of the agents on `USEbench` are shown in Table 3. We investigate the results per system.

`USEagent`. In general, `USEagent` demonstrates its applicability across different types of tasks in `USEbench`. `USEagent` achieves a PASS@1 of **45.6%** on **SWE-verified**. In comparison, AutoCodeRover, a more specialized agent on issue resolution, achieved efficacy of **46.2%** on the same SWE-verified benchmark when using the same Claude 3.5 Sonnet v2 backend LLM [1]. `USEagent` relaxed the fixed workflow in AutoCodeRover and is a more general agent that applies to more task types. Although being more general, `USEagent` demonstrates similar efficacy on the specialized issue resolution task compared to AutoCodeRover. This comparable performance demonstrates that, with the design of `USEagent`, the increased autonomy does not result in noticeable reduction in efficacy.

While being effective on issue resolution tasks, `USEagent` can handle other tasks which specialized agents could not be applied to. For issue reproduction tasks in **SWT**, `USEagent` resolved **40.3%** of the tasks. Here, "resolved" means `USEagent` generated test cases that cover all changed lines in the developer-written patch for the issue, without seeing the patch. Moreover, for a significant number

---

[2]at 95% confidence with 10% error margin.

[3]https://openrouter.ai/deepseek/deepseek-chat-v3-0324

**Table 3: Efficacy of USEagent and OpenHands CodeActAgent on USEbench.**

| Dataset | Tasks | USEagent | | | OpenHands | | |
|---|---|---|---|---|---|---|---|
| | | ☀PASS@1 | ☀³ PASS@5 | 🐳³ PASS@1 | ☀PASS@1 | ☀³ PASS@5 | 🐳³ PASS@1 |
| SWE-Ver | 500 | 228 (45.6%) | 66.7% | 42.7% | 192 (38.4%) | 52.1% | 35.9% |
| SWT | 298 | 120 (40.3%) | 53.6% | 34.8% | 85 (28.4%) | 50.7% | 29.0% |
| REPOCOD | 200 | 12 (6.0%) | 15.2% | 6.5% | 11 (5.5%) | 6.5% | 0% |
| REPOTEST | 173 | 55 (31.8%) | 42.5% | 17.5% | 45 (26.0%) | 42.5% | 10.0% |
| SWETRY | 100 | 8 (8.0%) | 30.4% | 8.7% | 7 (7.0%) | 8.7% | 4.3% |
| Total | 1271 | **423 (33.3%)** | 49.5% | 29.1% | **340 (26.8%)** | 44.1% | 22.6% |

of unresolved tasks, USEagent generated test cases that achieve a coverage above 90%. In these cases, the generated tests do not fully satisfy the requirement of the benchmark, but only miss coverage on individual statements or branches. On test generation tasks in **REPOTEST**, USEagent achieves efficacy of **31.8%**, which is slightly lower than similar tasks in SWT. While the requirement in SWT is to cover a developer-written patch, the requirement in REPOTEST is to generate tests that cover an entire method, which can be more challenging. Sometimes the agent generates tests that cover most scenarios in a method, but do not cover all the lines. On the other hand, we observe certain advantages of agent-generated tests compared to developer-written tests. The agent often generates shorter tests where each test has individual assertions, while the developer-written tests tend to consist of long tests that combine multiple inputs and assertions. In a real test-suite, short individual tests can provide more informative error messages when part of the test-suite fail [31, 32].

Tasks in **REPOCOD** proved to be the most challenging with only **6%** resolution rate. REPOCOD tasks are generally challenging because the requirement is to generate a complete method implementation that can pass a large number of hidden test cases. Overall, for many unresolved instances, the generated method passes a high number of tests, yet does not account for a few edge-cases. The majority of resolved instances stem from Sympy, a library for symbolic mathematics. We expect a relation between the relative success in Sympy and the fact that mathematical reasoning is becoming a common aspect of training in foundational models.

On the compound benchmark **SWETRY**, USEagent has a resolve rate of **8%**, which shows some promise to apply USEagent as a follow up to partial fixes. We need to stress that the data points in SWETRY are sourced from previous failed attempts of agentic systems, which means they are from the more challenging instances in SWE-verified. We observe two main failure modes on the SWETRY tasks. Firstly, since the partial patch is given to the agent as part of the task description, the agent often attempts to derive a solution on top of this partial patches by making modifications to it. However, the partial patch may be misplaced. In these cases, the agent still chose to iterate over this patch, and failed to discard the partial attempt and approach the task in a different way. Secondly, the `Meta-Agent` sometimes ends the agent execution prematurely. For example, when the `ExecuteTests` action still shows some relevant test errors, the `Meta-Agent` wrongly disregards errors as irrelevant to the task, ending the workflow. This can be addressed by configuring the prompt, at the cost of longer execution time.

OpenHands. OpenHands CodeActAgent achieves a PASS@1 of **38.4%** on **SWE-verified**, which is lower than the reported result from the SWE-Bench leaderboard [1]. We attribute this difference to the choice of agent - the reported result on the leaderboard is generated using a specialized `CodeActSWEAgent`, while we used the more general `CodeActAgent` in our evaluation.

We utilize the general `CodeActAgent` to solve the different types of tasks in USEbench. For **SWT**, `CodeActAgent` achieves a resolution rate of a **28.4%**, slightly lagging behind USEagent. Similar to that of USEagent, `CodeActAgent` only achieves a low resolution rate on **REPOCOD** (5.5%). However, unlike USEagent, resolved instances are evenly distributed across `sympy`, `astropy` (a popular library for Astronomy and Astrophysics), and `Plotly` (a graphing library), each contributing 27% of the total resolved cases. On **REPOTEST**, OpenHands CodeActAgent resolves 26.0% of the total instances, which is similar but slightly lower compared to its performance on SWT. This trend is consistent with that of USEagent. On **SWETRY**, `CodeActAgent` only solves 7%, similar to that of USEagent, indicating that improving partial fixes is still a difficult task for general agents.

*Open-Source Model.* We also report the efficacy of the agents with the open-source model DeepSeek-V3 in Table 3. With DeepSeek-V3, both USEagent and OpenHands have a slight performance drop, which is expected. There is a larger performance drop on REPOTEST for both agents. REPOTEST requires generating a large amount of test code to achieve full test coverage – DeepSeek-V3 sometimes makes minor mistakes in the generated code, or generates tests that only partially cover the target function.

*PASS@5.* Moving to a PASS@5 increases the efficacy of USEagent to 49.5% (**+48.6%**) and OpenHands to 44.1% (**+64.6%**). This shows a similar increase across the two systems when employing retries.

> **Summary: RQ1 - Efficacy of Agentic Systems**
> USEagent solves the tasks from USEbench at a 33.3% rate, compared to 26.8% of OpenHands. There is a consistent improvement over all benchmarks. USEagent's efficacy of 45.6% for SWE-verified is close to the performance of fine-tuned systems.

## 6.2 RQ2: Dissection of Plausible Patches

For USEagent, we manually investigated a sample of 218 plausible solutions (i.e. passing the resolution criteria of each benchmark) to

---

³Derived from the statistically significant subset

**Table 4: Results of ablation study with USEagent on a statistically significant subset of USEbench, using claude-3-5-sonnet-20241022 as the backend model.**

| Dataset | USEagent original | USEagent disable dynamic | USEagent remove ExecuteTests |
|---|---|---|---|
| SWE-Ver | 49.6% | 46.2% (-7%) | 49.6% (-0%) |
| SWT | 40.6% | 39.1% (-4%) | 36.2% (-11%) |
| REPOCOD | 8.7% | 6.5% (-25%) | 0% (-100%) |
| REPOTEST | 25.0% | 20.0% (-20%) | 17.5% (-30%) |
| SWETRY | 13.0% | 8.7% (-33%) | 8.7% (-33%) |
| Overall | 34.9% | 31.9% (-9%) | 31.2% (-11%) |

understand the degree of memorization, overfitting or other anomalies. We report a low number of 3 cases of memorization (**1.3%**), next to 23 cases of overfitting (**10.5%**). The percentage of overfitting solutions is lower than the previously reported overfitting rate of 31% on SWE-bench from SpecRover [29]. We attribute the lower overfitting rate in part to our extensive ExecuteTests action in USEagent, which allows for more versatile test execution. As a result, we see an overfitting rate of 13.6% from USEagent on the SWE-verified dataset. In addition, the SWT and REPOTEST datasets show low numbers of overfitting, which further reduces the overall rate. Furthermore, we identified a set of 33 anomalies that are plausible, and do not overfit, yet they are different from the ground truth. One example anomaly includes the introduction of a new vector library for xarray (a vector library itself). Another example is an unorthodox class-inheritance check for scikit-learn - instead of utilizing pythons standard function isinstance, a predicate is applied to the objects obj.__classes__. These solutions are functionally correct but are unlikely to be accepted by developers, so we have marked them as anomalies.

> **Summary: RQ2 - Analysis of Plausible Solutions**
> We manually inspected 218 plausible solutions from USEagent and report 10.5% overfitting, 1.3% memorization. Overfitting was most common within SWE-verified, and least common in REPOTEST and SWT.

## 6.3 RQ3: Self-Configuration

Figure 4 shows the different choices of the Meta-Agent when facing different tasks (we present two for brevity). The histograms illustrate the actions taken at each step in the sequence (first, second, etc.). We observe some clear patterns in the different task types that match our intuition. For program repair tasks (i.e. in SWE-Verified), we see in Figure 4a that the first step prominently consists of generating a reproduction test, followed by retrieving relevant code context. Over the course of agent execution, the EditCode and ExecuteTests become more prominent, introducing and verifying code changes to the software. For regression testing (in SWT), Figure 4b shows that by-large the first step chosen is TestRetrieval, immediately followed by a (test-)code edit. The majority of USEagent trajectories on SWT tasks then consist of alternating test-changes and test-executions. Overall, the sequence of action invocation follows a certain pattern for each task type as shown in Figure 4, showing that USEagent can self-configure its workflow based on the task type. Across all benchmarks, USEagent

first gathers information (using the retrieval actions or reproducers), converging towards alternating EditCode and ExecuteTests until either budgets are exhausted or Terminate is chosen. This behavior showcases the capability of emulating concepts like Test-Time Scaling [9] and command-discovery [8] through the Meta-Agent.

Table 4 supports these findings through an ablation study. The *disable dynamic* column refers to USEagent with the dynamic decision making by MetaAgent disabled. Instead of letting the MetaAgent freely decide which action should be taken as the next step, we predefine the following workflows for the task types:

- SWE/SWETRY: Reproduction -> CodeRetrieval -> EditCode -> (ReviewPatch -> EditCode)*
- SWT: TestRetrieval -> CodeRetrieval -> EditCode -> (ExecuteTests -> EditCode)*
- REPOCOD: EditCode -> TestRetrieval -> EditCode -> (ExecuteTests -> EditCode)*
- REPOTEST: TestRetrieval -> EditCode -> (ExecuteTests -> EditCode)*

where * represents a loop, and the agent can decide when to break out the loop and end the workflow. We see that disabling dynamic decision making leads to reductions in efficacy from 4% to 33% across tasks. The MetaAgent in USEagent not only allows for it to handle multiple task types without hurting the efficacy compared to predefined workflows, but without it there would be an efficacy drop. This efficacy improvement (by the MetaAgent) is largely because the MetaAgent can help to recover from initial sub-optimal decisions (e.g. invoke retrieval actions towards the end of workflow to gather more context).

We performed another ablation study to examine the effect of an individual action. The column *remove ExecuteTests* in Table 4 shows the efficacy of USEagent after removing ExecuteTests, which is one of the actions newly introduced in this work. In task types that require repeated test executions to validate and refine solutions (e.g. REPOCOD and REPOTEST), removing ExecuteTests results in significant efficacy drops (-100% and -30%). On the other hand, there is no efficacy drop in SWE-bench tasks, since the MetaAgent could invoke alternative actions such as ReviewPatch to refine solutions. These results indicate that a single action can enhance effectiveness across multiple task types.

> **Summary: RQ3 - Self Configuration**
> USEagent is capable of choosing different *action-patterns* for different benchmarks. The self-configured workflow is as effective as predefined static workflows. In addition, a single action can help improve efficacy in multiple task types.

## 6.4 RQ4: Open Challenges in Agentic Systems

Based on our investigations, we identify several major challenges in the current unified software engineering agent. Firstly, when the task requires writing a large amount of code (e.g. feature development tasks), the agent often generates solutions that are "mostly correct". This is reflected on our evaluation on the REPOCOD dataset, where the task is to implement a complex function based on natural language requirement, and the correctness is defined as passing a large number of hidden unit tests. We observe a number of solutions generated by the agent implement the required feature almost
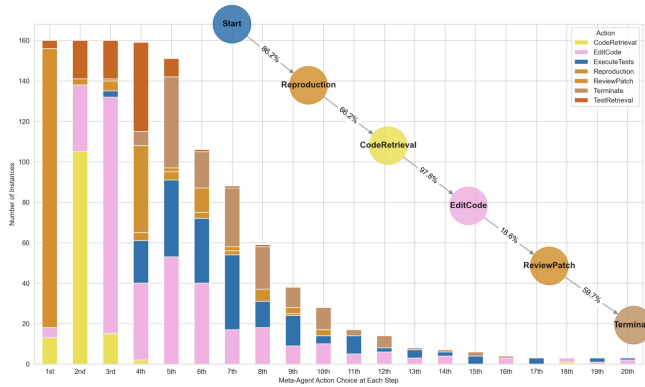
(a) Distribution of SWE Steps – the most common trajectory begins with reproduction, followed by code retrieval, editing, and review.
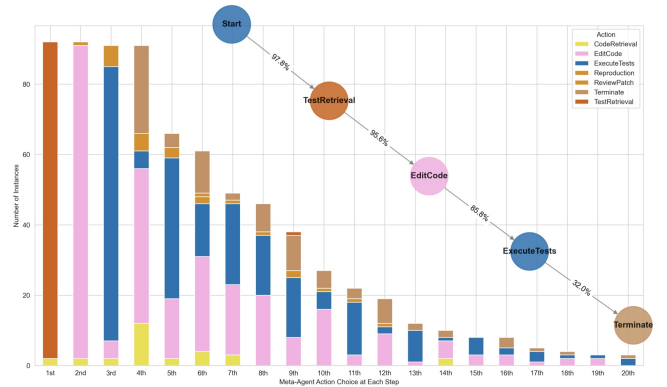
(b) Distribution of SWT Steps - the most common trajectory begins with test-case retrieval, followed by test-editing and execution.

**Figure 4: Comparison of SWT and SWE Step Distributions for `USEagent` on a significant subset of `USEbench`. Each bar shows how often different actions are invoked at a particular `Meta-Agent` step.**

correctly, but fail a few hidden tests due to missing edge cases. The edge-case behaviors are sometimes not specified clearly in the natural-language task description. We attribute this challenge in part to the inherent ambiguity of natural language. A future agent may first resolve the ambiguity in natural language by transforming the task requirements into more "formal" artifacts such as specifications and test cases, and then generate solutions based on the formal specification. Another approach is to design human-agent interaction schemes to clarify ambiguity when needed.

Secondly, the current agent lacks the capability of "backtracking" when the agent execution does not yield meaningful results on a given execution path. For example, when there is a partial patch at a particular program location, the agent is prone to continuously make small edits to the partial patch and execute tests to verify them. However, the agent may not make good progress with this approach, since it may be impossible to craft a good solution at the location of the partial patch. This issue is amplified in the SWETRY dataset, in which the task description comes with a partial patch. To tackle this issue, a future agent should employ a backtracking mechanism to discard unpromising partial solutions, and start over at a previous step. It could be possible to employ the recent reasoning LLMs to examine the agent execution trace and decide whether to backtrack, since reasoning models have shown backtracking behaviors in their thinking process [16]. Moreover, recent work [5] on employing search algorithm over agent execution trajectories can help the agent escape from unpromising paths.

Lastly, we still observe patch *overfitting* in tasks such as program repair and regression testing. Overfitting is a known problem for automated program repair [15]. Overfitting happens due to the generated patches passing given tests, but missing actual requirements. Thus, to avoid overfitting, if the tests are too specific, we need to generalize them. Agentic systems show promise for test generation (also reflected in this work), and the `USEagent` framework allows additional actions, such as test amplification or mutation testing, to be incorporated into the agent. Artifacts from these actions can be fed back into code generation, thereby reducing overfitting and enhancing trustworthiness.

---

**Summary: RQ4 - Open Challenges**

Challenges include exploring edge-case handling and alternative solutions in feature development, backtracking and discarding poor partial results in repair, and addressing patch overfitting.

---

## 7 RELATED WORK

*SE - Benchmarks.* Recent Software Engineering Benchmarks fall into two broad categories: isolated coding exercises and repository-level tasks. Some coding datasets such as CodeXGlue [22] come without an evaluation metric and form a training corpus; others such as HumanEval [11], MBPP [6], ClassEval [13] or CoderEval [40] provide a challenge accompanied by an evaluation harness (i.e., a test-suite). There are already ongoing efforts to provide a meta-benchmark for these tasks [21, 41], which commonly target LLMs directly rather than agentic systems. For `USEbench`, we select repository-level tasks to ensure challenges of uniform granularity. We opted against coding exercises, as many models have begun to saturate on such benchmarks [30].

*LLM Agents for Software Engineering.* We consider a system *agentic* if the LLM exhibits a degree of autonomy and can interact with the environment through commands or other pre-defined tools. Within agentic systems, we distinguish terminal-based agents that interface LLMs directly with a console and minimal tooling (e.g., SWE-Agent [38], Google Jules [14]), from approaches that integrate additional domain knowledge and techniques. These additions include specialized tools (such as spectrum-based fault localization in AutoCodeRover [42]), self-reflective capabilities (such as the reviewer agent in SpecRover [29]), and control logic (such as the finite-state machine in RepairAgent [7]). In this work, we have focused on AutoCodeRover [42] and OpenHands [35], as both are widely used open-source projects. While the Aider project [3] covers a well-suited scope involving multiple types of software engineering tasks, its workflow relies on human prompting. Passerine [28] by Google proposes a program repair framework inspired by SWE-Agent [38] and incorporates a structure similar to our Meta-Agent, providing five pre-configured commands (and does not allow free execution

**Table 5: Comparing USEagent with LLM agents for software engineering. All these agents take in natural language issues.**

| | USEagent | SWE-Agent | OpenHands CodeAct | AutoCode Rover | Agentless | MASAI | Passerine | CodeR |
|---|---|---|---|---|---|---|---|---|
| Is workflow predefined or dynamically constructed? | dynamic | dynamic | dynamic | predefined | predefined | predefined | dynamic | predefined |
| Handles only program repair or multiple task types? | multiple | multiple | multiple | program repair | program repair | program repair | program repair | program repair |
| What is the "unit of work" in the agent? | SE action | terminal-level command | terminal-level command | SE action | SE action | SE action | SE action | SE action |
| Multi-agent system? | yes | no | no | yes | no | yes | no | yes |
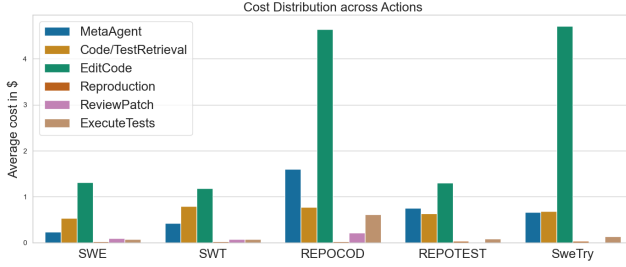


**Figure 5: Average cost across Actions in USEagent.**

of terminal commands, unlike SWE-Agent). Our approach differs in scope, as we target a broader range of tasks beyond program repair. MASAI [33] defines an action space for its central agent similar to our Meta-Agent, but is also limited to program repair. In contrast, we structure the essential artifacts involved in software debugging via a task state, incorporate a broader set of software engineering actions for orchestration, and apply the agentic framework to multiple task types. CodeR [10] has a Meta-Agent component that pre-defines an execution plan, but does not adapt it dynamically as USEagent does. Table 5 compares USEagent with existing software engineering agents that take in natural language issues. USEagent distinguishes itself from the rest by dynamically orchestrating actions (each action is executed by sub-agents) to handle multiple types of software engineering tasks.

## 8 THREATS TO VALIDITY

*Construct Validity.* Within USEbench, we approximate testing capabilities using test coverage. While test coverage does not guarantee semantic coverage and can sometimes be achieved through various *hacks*, it remains a useful proxy in the absence of a better oracle. It captures several attributes we aim to evaluate: code changes must be correctly formatted and target the appropriate program locations. Moreover, upon inspection, many of the REPOTEST datapoints contain branches that handle edge-case behavior with errors, where syntactic coverage effectively reflects semantic testing.

*Internal Validity: Data Leakage.* It is possible that LLMs have been exposed to the projects under evaluation or even datapoints from the source benchmarks [30]. While fully preventing such exposure is beyond our capabilities and the scope of this work, we aim to provide an overview of potential memorization by manually inspecting a sample of plausible solutions.

## 9 DISCUSSION

*Cost Analysis.* With Claude 3.5 Sonnet v2 as the backend model, USEagent costs were at an average of 3.65 USD, with notable differences across sub-benchmarks. Herein SWE scored lowest with 2.20 USD average, and REPOCOD highest at a mean of 7.66 USD. The high costs in REPOCOD originate from the iteration of patch generation and test execution. As shown in Figure 5, among all actions, EditCode is the main cost-driver, due to the large contexts provided through relevant code and tests. We calculate the Pearson correlation coefficient for the elapsed time and the costs, and find a strong correlation of $r = 0.89$, implying correlation between computational (e.g. test execution) and model (e.g. reasoning) efforts. When using DeepSeek-V3 as backend model, the average cost of USEagent is 0.55 USD per datapoint, which is ~15% of the cost with Claude 3.5 Sonnet v2. Given the efficacy presented in Table 3, open-source models are cost-effective.

*Perspectives.* We consider the approach of USEagent to be a promising avenue for solving *any* task that revolves around code as its primary artifact. Many tasks such as dependency management or project configuration can be incrementally tackled by introducing more actions into USEagent. For USEagent to become truly an *AI Software Engineer*, it must address tasks such as requirements engineering, data visualization, deployment, code review or even a-b-testing. The efforts presented in this work are the second step in agentic development towards an AI software engineer, the first being individual agentic systems for single tasks (like AutoCodeRover). With the USEagent we pave the way for a unified approach to interact autonomously with code regardless of the task. Going beyond the USEagent will involve studying the cooperative intelligence resulting from multiple USEagents and multiple human developers. This might establish the dynamics of future development teams.

*Artifact Availability.* We release the logs and evaluation of USEagent at https://doi.org/10.5281/zenodo.15023393, the source code of USEbench at https://github.com/nus-apr/USEbench, and the source code of USEagent at https://github.com/nus-apr/USEagent.

# REFERENCES

[1] [n. d.]. SWE-bench leaderboard. https://www.swebench.com/#verified. Accessed: 2025-03-08.

[2] M Pradel A Roychoudhury, C Pasareanu and B Ray. 2025. AI Software Engineer: Programming with Trust. *arxiv* (Feb 2025).

[3] AIDER Team. 2024. AIDER: Advanced AI-Powered Coding Assistant. https://aider.chat. Accessed: 2025-01-03.

[4] Anonymous. 2021. The Big Code Project: Learning to Program by Learning to Understand Programming Languages. *arXiv preprint arXiv:2102.01092* (2021).

[5] Antonis Antoniades, Albert Örwall, Kexun Zhang, Yuxi Xie, Anirudh Goyal, and William Wang. 2024. SWE-Search: Enhancing Software Agents with Monte Carlo Tree Search and Iterative Refinement. *arXiv preprint arXiv:2410.20285* (2024).

[6] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

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

[8] Islem Bouzenia and Michael Pradel. 2024. You Name It, I Run It: An LLM Agent to Execute Tests of Arbitrary Projects. *arXiv preprint arXiv:2412.10133* (2024).

[9] Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V Le, Christopher Ré, and Azalia Mirhoseini. 2024. Large language monkeys: Scaling inference compute with repeated sampling. *arXiv preprint arXiv:2407.21787* (2024).

[10] Dong Chen, Shaoxin Lin, Muhan Zeng, Daoguang Zan, Jian-Gang Wang, Anton Cheshkov, Jun Sun, Hao Yu, Guoliang Dong, Artem Aliev, et al. 2024. CodeR: Issue Resolving with Multi-Agent and Task Graphs. *arXiv preprint arXiv:2406.01304* (2024).

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. (2021). arXiv:2107.03374 [cs.LG]

[12] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Ryan Puri, Jared Nixon, Pamela Mishkin, Daniel Nguyen, Jordan Kaplan, and Ananthan Rajasekaran. 2021. Evaluating Large Language Models Trained on Code. *arXiv preprint arXiv:2107.03374* (2021).

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

[14] Google. 2024. Google Gemini AI Update: December 2024. Blog post. https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/#building-responsibly Available online: https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/#building-responsibly.

[15] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *Commun. ACM* 62 (2019). Issue 12.

[16] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[17] Shanshan Han, Qifan Zhang, Yuhang Yao, Weizhao Jin, Zhaozhuo Xu, and Chaoyang He. 2024. LLM multi-agent systems: Challenges and open problems. *arXiv preprint arXiv:2402.03578* (2024).

[18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[19] René Just, Darioush Jalali, and Michael D Ernst. 2014. 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*. 437–440.

[20] Shanchao Liang, Yiran Hu, Nan Jiang, and Lin Tan. 2024. Can Language Models Replace Programmers? REPOCOD Says' Not Yet'. *arXiv preprint arXiv:2410.21647* (2024).

[21] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2024. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems* 36 (2024).

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

[23] Niels Mündler, Mark Niklas Mueller, Jingxuan He, and Martin Vechev. 2024. SWT-Bench: Testing and Validating Real-World Bug-Fixes with Code Agents. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=9Y8zUO11EQ

[24] OpenAI. 2024. Introducing SWE Bench: Verified. https://openai.com/index/introducing-swe-bench-verified/. Accessed: 2025-01-03.

[25] OpenAI. 2024. OpenAI o1: Large Language Model. https://openai.com/index/introducing-openai-o1-preview/ Accessed: 2025-02-26.

[26] Andrew Orwall and contributors. 2025. Moatless Tools. https://github.com/aorwall/moatless-tools/tree/main. Accessed: 2025-02-04.

[27] Ruchir Puri, Geert Jansen Saini, Shruti Kalyan, Naveen Limaye, Saurabh Kolar, Philipe Maldonado, Jianfeng Hu, Dinesh Sreedhar, Yunhui Zhang, Lingming Zhang, et al. 2021. CodeNet: A Large-Scale AI for Code Dataset for Learning a Diversity of Coding Tasks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 15–18.

[28] Pat Rondon, Renyao Wei, José Cambronero, Jürgen Cito, Aaron Sun, Siddhant Sanyam, Michele Tufano, and Satish Chandra. 2025. Evaluating Agent-based Program Repair at Google. *arXiv preprint arXiv:2501.07531* (2025).

[29] Haifeng Ruan, Yuntong Zhang, and Abhik Roychoudhury. 2024. Specrover: Code intent extraction via llms. *arXiv preprint arXiv:2408.02232* (2024).

[30] June Sallou, Thomas Durieux, and Annibale Panichella. 2024. Breaking the silence: the threats of using llms in software engineering. In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results*. 102–106.

[31] Davide Spadini, Fabio Palomba, Andy Zaidman, Magiel Bruntink, and Alberto Bacchelli. 2018. On the relation of test smells to software code quality. In *2018 IEEE international conference on software maintenance and evolution (ICSME)*. IEEE, 1–12.

[32] Arie Van Deursen, Leon Moonen, Alex Van Den Bergh, and Gerard Kok. 2001. Refactoring test code. In *Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001)*. Citeseer, 92–95.

[33] Nalin Wadhwa, Atharv Sonwane, Daman Arora, Abhav Mehrotra, Saiteja Utpala, Ramakrishna B Bairi, Aditya Kanade, and Nagarajan Natarajan. 2024. MASAI: Modular Architecture for Software-engineering AI Agents. In *NeurIPS 2024 Workshop on Open-World Agents*. https://openreview.net/forum?id=NSINt8lLYB

[34] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *ICML*. arXiv:2402.01030

[35] Xingyao Wang, Boxuan Li, Yufan Song, Frank F Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. 2024. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741* (2024).

[36] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *IEEE Transactions on Software Engineering* 42, 8 (2016), 707–740.

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

[38] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793* (2024).

[39] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2023. ReAct: Synergizing Reasoning and Acting in Language Models. In *International Conference on Learning Representations (ICLR)*.

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

[41] Zhaojian Yu, Yilun Zhao, Arman Cohan, and Xiao-Ping Zhang. 2024. HumanEval Pro and MBPP Pro: Evaluating Large Language Models on Self-invoking Code Generation. *arXiv preprint arXiv:2412.21199* (2024).

[42] Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. 2024. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1592–1604.

[43] Zeyu Zhang, Xiaohe Bo, Chen Ma, Rui Li, Xu Chen, Quanyu Dai, Jieming Zhu, Zhenhua Dong, and Ji-Rong Wen. 2024. A survey on the memory mechanism of large language model based agents. *arXiv preprint arXiv:2404.13501* (2024).