SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering

John Yang[*1] Carlos E. Jimenez[*1] Alexander Wettig[1]
Kilian Lieret[1] Shunyu Yao[1] Karthik Narasimhan[1] Ofir Press[1]
[1]Princeton Language and Intelligence (PLI), Princeton University

Abstract

Software engineering is a challenging task requiring proficiency in both code generation and interacting with computers. In this paper, we introduce SWE-agent, an autonomous system that uses a language model to interact with a computer to solve software engineering tasks. We show that a custom-built agent-computer interface (ACI) greatly enhances the ability of an agent to create and edit code files, navigate entire repositories and execute programs. On SWE-bench, SWE-agent is able to solve 12.5% of issues, compared to the previous best of 3.8% achieved with retrieval-augmented generation (RAG). We explore how ACI design impacts an agent's behavior and performance, and provide insights on effective design.

1 Introduction

Language models (LMs) have become helpful assistants for software development (Chen et al., 2021; Austin et al., 2021; Li et al., 2022), with users playing the role of mediator between the LM and the computer to complete programming tasks. For instance, after executing LM-generated code, a user may request refinements from the LM based on computer feedback such as error messages. More recently, LMs have been used as autonomous *agents* that interact with computer environments without human intervention (Yang et al., 2023a; Wu et al., 2024; Xie et al., 2024). This approach has the potential to accelerate software development, but remains largely unexplored in realistic settings.

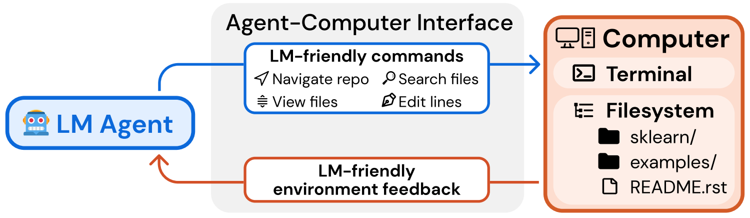


Figure 1: An agent interacts with a computer through an agent-computer interface (ACI), which includes the commands the agent uses and the format of the feedback from the computer.

This work introduces SWE-agent, an LM-based autonomous system that can interact with a computer to solve challenging real-world software engineering problems from SWE-bench (Jimenez et al., 2024). At every turn, SWE-agent outputs a thought and a command, and then receives feedback from the execution of the command in the environment (ReAct; Yao et al. (2023b)). As our main contribution, we establish the importance of designing an *agent-computer interface* (ACI) to enhance the LM agent's performance. We find that custom ACIs tailored to LMs can outperform existing user interfaces (UIs), such as the Linux shell, designed for human users.

Consider the baseline of using a Linux shell interface (Yang et al., 2023a), which naturally lends itself to turn-based interaction with an LM agent due to its text-only input commands and outputs.

[*]Equal contribution. Correspondence to {`jy1682`, `carlosej`}`@princeton.edu`.
Code and demo at swe-agent.com.

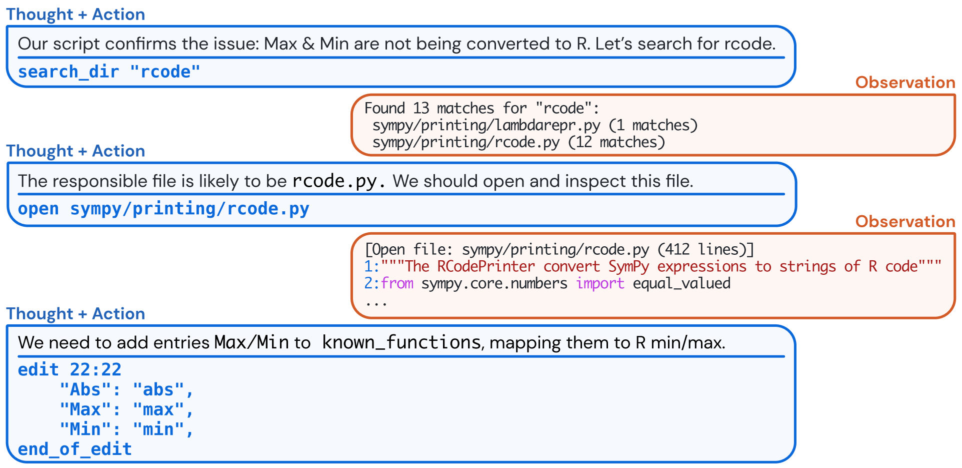
In practice, we find that this is an unsuitable environment for agents. For example, it does not provide simple commands to edit a small chunk of a file, and does not provide any feedback when invalid edits are made (Figure 5, left). We find that this substantially hampers performance (Section 5). This leads us to construct a novel agent-computer interface for SWE-agent, which provides commands for viewing, searching through and editing files, along with carefully crafted environment feedback (Figure 1), including informative error messages for edits that contain syntax errors (Figure 5, right). We empirically show that these ACI elements substantially improve performance.

When using GPT-4 Turbo as the base LLM, SWE-agent solves 12.5% of the 2,294 SWE-bench test issues, substantially outperforming the previous best resolve rate of 3.8% by a non-interactive, retrieval-augmented system. We perform an ablation study on a subset of 300 SWE-bench test issues to analyze our ACI design choices (Table 2), and show that SWE-agent solves 10.7 percentage points *more* issues than the baseline agent that uses just the default Linux shell. Even though our ACI was developed for GPT-4 Turbo, we find the interface transfers to a different LM, namely, SWE-agent with Claude 3 Opus can solve 10.5% issues.

To summarize, our contributions are twofold: first, we build, evaluate, and open-source SWE-agent, a state-of-the-art system that can solve real-world software engineering tasks, and second, we introduce the concept of the agent-computer interface (ACI) and show how ACI design can lead to substantial improvements in the performance of LM agents, without modifying the weights of the underlying language model. We hope our work can drive the community towards further research on autonomous LM agents and agent-computer interfaces (ACIs).

2 The Agent-Computer Interface

An LM acts as an agent by interacting with an environment through iteratively taking actions and receiving feedback (Yao et al., 2023b; Sumers et al., 2023). Typically the environment is fixed, such as in robotics, where agents control actuators in the physical world. However, digital environments offer the flexibility of developing an interface between the agent and the computer, reflected by the wide variety of interfaces for programs (application programming interfaces, APIs) and humans (user interfaces, UIs). Here, we argue that LM agents represent a new category of end users and we call the interface they use to interact with computers the *agent-computer interface* (ACI).



Our script confirms the issue: Max & Min are not being converted to R. Let's search for rcode.

```
search_dir "rcode"
```

```
Found 13 matches for "rcode":
 sympy/printing/lambdarepr.py (1 matches)
 sympy/printing/rcode.py (12 matches)
```

The responsible file is likely to be `rcode.py`. We should open and inspect this file.

```
open sympy/printing/rcode.py
```

```
[Open file: sympy/printing/rcode.py (412 lines)]
1:"""The RCodePrinter convert SymPy expressions to strings of R code"""
2:from sympy.core.numbers import equal_valued
...
```

We need to add entries `Max/Min` to `known_functions`, mapping them to R min/max.

```
edit 22:22
    "Abs": "abs",
    "Max": "max",
    "Min": "min",
end_of_edit
```

Figure 2: This trajectory snippet shows how SWE-agent interacts with the computer using the ACI. The LM-generated thoughts and actions are shown in blue, and the computer feedback in red.

Agents interact with a computer in a series of turns: iterating between the agent issuing commands and the computer responding with the command's output, visualized in Figure 2. The ACI specifies the commands available to the LM and how the environment state after the execution of each command will be communicated back to the LM. It also tracks the history of all previous commands and observations, and at each step, manages how these should be formatted and combined with high-level instructions into a single input for the language model.

In this paper, we keep the language model fixed, and we focus on designing the ACI to achieve better performance. This means that we shape the available actions, their documentation, and the environment feedback to complement a language model's limitations and abilities. We use two approaches to enhance the ACI on a development set: We manually inspect the behaviour of the agent to identify difficulties and propose improvements, and we run a grid search over ACI configurations.

This leads us to several observations on what makes for effective ACIs:

1. **Actions should be simple and easy to understand.** Many bash commands have documentation that includes dozens of options. Simple commands with a few options and concise documentation are easier for agents to use, reducing the need for demonstrations or fine-tuning. This is a defining principle for all SWE-agent commands covered in Section 3.
2. **Actions should be efficient.** Important operations (e.g., file navigation, editing) should be consolidated into as few actions as possible. Efficient actions help agents make meaningful progress towards a goal in a single step. A bad design is to have many simple actions that need to be composed across multiple turns for a higher order operation to take effect. We incorporate this into the Editing and Search interface designs covered in Section 3 and analyzed in Section 5.1.
3. **Environment feedback should be informative.** High quality feedback provides the agent with substantive information about the current environment state (and the effect of the agent's recent actions) without providing unnecessary details. For instance, when editing a file, updating the agent on the revised contents is helpful. Figures 3a, 3b and Table 2 provide evidence for this.
4. **Guardrails mitigate error propagation.** Like humans, LMs can make mistakes when editing or searching, but LMs struggle to recover from these errors. Building in guardrails, such as a code syntax checker, that automatically detects mistakes, can help agents recognize errors and then fix them. We show the effect of editing guardrails in Table 2.

We provide analysis and ablation studies in Section 5 to show how alternative interfaces affect performance.

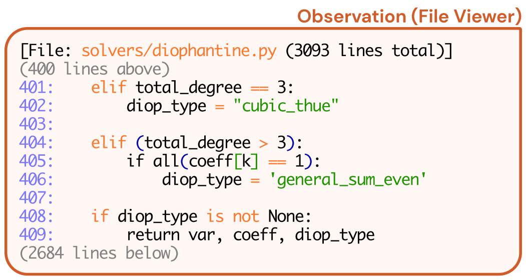
3 SWE-agent: Designing an ACI for Software Engineering

SWE-agent uses an ACI that enables an LM to act as a software engineering agent. To evaluate our system, we use SWE-bench, a benchmark for evaluating LMs on real world software issues collected from GitHub (Jimenez et al., 2024). SWE-bench's issue resolution task reflects an important responsibility of software engineers: Given a codebase and a natural language request (e.g. feature request, bug report), generate a codebase revision (i.e. a patch) that passes unit tests which verify that the issue has been remedied. Solving these tasks requires performing a number of software engineering sub-tasks, including bug localization, program repair, and writing tests.

Existing application interfaces to perform software engineering have naturally been designed around the human user. For instance, developers frequently use GUI-based IDEs, such as VSCode, or screen-based text editors, such as `Vim`. These interfaces have rich visual components and feedback that make them powerful tools for humans, but may not be as suited to LMs.

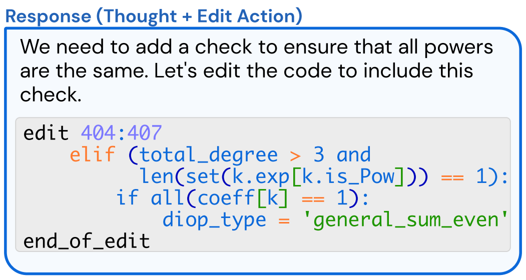
Consider a text editor like `Vim` which relies on cursor-based line navigation and editing. Carrying out any operation leads to long chains of granular and inefficient interactions. Furthermore, humans can ignore unexpected inputs, such as accidentally outputting a binary file or thousands of lines returned from a 'grep' search. LMs are sensitive to these types of inputs, which can be distracting and take up a lot of the limited context available to the model. On the other end, commands that succeed silently confuses LMs. We observe that LMs will often expend extra actions to verify that a file was removed or an edit was applied if no automatic confirmation is given. We show that LMs are substantially more effective when using interfaces built with their needs and limitations in mind.

SWE-agent overcomes these challenges by introducing powerful, tailored tools to provide agents with a simplified interface to the computer; i.e. the agent-computer interface (ACI). A well-designed ACI should help the agent understand the state of the repository given the previous changes. It should also help the agent recover from mistakes, remove unnecessary context from prior observations, and suppress any unusual, lengthy program outputs.



```
[File: solvers/diophantine.py (3093 lines total)]
(400 lines above)
401:     elif total_degree == 3:
402:         diop_type = "cubic_thue"
403:
404:     elif (total_degree > 3):
405:         if all(coeff[k] == 1):
406:             diop_type = 'general_sum_even'
407:
408:     if diop_type is not None:
409:         return var, coeff, diop_type
(2684 lines below)
```

(a) Observation from the file viewer.



We need to add a check to ensure that all powers are the same. Let's edit the code to include this check.

```
edit 404:407
    elif (total_degree > 3 and
          len(set(k.exp[k.is_Pow])) == 1):
        if all(coeff[k] == 1):
            diop_type = 'general_sum_even'
end_of_edit
```

(b) Action using the edit interface.

Figure 3: The file viewer and edit command are integrated with each other. Here, in (a), the file viewer shows the agent the contents of the open file with line numbers , and in (b), the agent invokes the edit function to replace lines 404-407 in the open file. After the edit, the file viewer will show the agent the updated version of the file again.

SWE-agent provides an ACI for LMs to act as software engineering agents, allowing them to effectively search, navigate, edit, and execute code commands. SWE-agent does this through the careful design of the agent's **search / navigation**, **file viewer**, **file editor**, and **context management**. Our system is built on top of the Linux shell and allows access to common Linux commands and utilities. We describe the components of the SWE-agent interface in greater detail below.

Search / navigation. Navigating codebases requires finding the right file and content that is relevant to your purposes. A common strategy for this is to look up terms that might be useful, like files, functions, or class definitions mentioned in an issue. We introduce special commands `find_file`, `search_file`, and `search_dir` that output a summary of search results when searching for filenames and strings within files or directories. Examples of these search result formats can be found in Figures 8. The `find_file` command can be used to search for filenames in the repository, while the `search_file` and `search_dir` are used to locate strings in a file or the files of a subdirectory respectively. Our interface encourages efficient searches by suppressing verbose results. The search commands return at most 50 results for each search query. If a search produces more than 50 results, we do not report the results and instead suggest to the agent to write a more specific query.

File viewer. Once models have found a file they want to view, they can do so using the interactive file viewer by calling the command `open` on the file path they want to view. The file viewer presents a window of at most 100 lines of the file at a time. The agent can move this window with the commands `scroll_down` and `scroll_up` or skip to a specific line with the `goto` command. To facilitate in-file navigation and code localization we display the full path of the open file, the total number of lines in the file, the number of lines omitted before and after the current window, and the line number (prepended to each line visible). An example of this interface is shown in Figure 3a.

File editor. We provide a few commands that allow models to create and edit files. The `edit` command works in conjunction with the file viewer, allowing agents to replace a specific range of lines in the open file. The `edit` command takes 3 required arguments: the start line, end line, and replacement text. In a single step, agents can replace all lines between the start and end lines with the replacement text, which is visualized in Figure 3b. After edits are applied, the file viewer automatically displays the updated content, allowing the agent to observe the effects of its edit immediately without needing to invoke any additional commands. An example agent response including an file edit is shown in Figure 3b.

Similar to how humans may use tools like syntax highlighting to alert them of format errors when editing files in an IDE, we integrate a code linter into the `edit` function to alert the model of mistakes it may have introduced when editing a file. Errors from the linter are shown to the model with a snippet of the file contents before and after the error was introduced. Invalid edits are discarded and the model is asked to try editing the file again.

Context management. The SWE-agent system uses informative prompts, error messages, and history processors to keep agent context concise and informative. Agents receive instructions, documentation, and a demonstrations on the correct use of bash and ACI commands. At each step, agents are instructed to generate both a *thought* and an *action* (Yao et al., 2023b). Malformed generations trigger an error response, shown in Figure 15, asking the model to try again, which is repeated until a valid generation is received. Once a valid generation is received, past error messages are omitted except for the first. The agent's environment responses display computer output using the template shown in Figure 14, but if no output is generated, a specific message stating "Your command ran successfully and did not produce any output" is included to enhances clarity. To further improve context relevance, observations preceding the last 5 are each collapsed into a single line. By removing most of the content from prior observations, we maintain essential information about the plan and action history while reducing unnecessary content, which allows for more interaction cycles and avoids showing outdated file content. More on the implementation is provided in Appendix A.

4 Experimental Setup

Datasets. We evaluate on the SWE-bench dataset, which includes 2,294 task instances from 12 different repositories of popular Python packages (Jimenez et al., 2024). We report our main agent results on the full SWE-bench test set and ablations and analysis on the SWE-bench Lite test set, unless otherwise specified. SWE-bench Lite is a canonical subset of 300 instances from SWE-bench with a focus on evaluating self-contained functional bug fixes.

Models. All results, ablations, and analysis are based on two leading LMs, GPT-4 Turbo (`gpt-4-1106-preview`) (OpenAI et al., 2023) and Claude 3 Opus (`claude-3-opus-20240229`) (Chiang et al., 2024). We experimented with a number of additional closed and open source models, including Llama 3 and DeepSeek Coder, but found their performance in the agent setting to be subpar. Many LMs' context window is too small, such as Llama 3 with a context window of 8k. GPT-4 Turbo and Claude 3 Opus have 128k and 200k token context windows respectively,[1] which provides enough room for the LM to interact for several turns after being fed the initial set of system, issue description, and optionally, demonstrations.

Baselines. We compare SWE-agent against two baselines. The first setting is the non-interactive, retrieval-augmented generation (RAG) baselines established in Jimenez et al. (2024). Here, a BM25 retrieval system is used to retrieve the most relevant codebase files using the issue as the query. Provided these files, the model is asked to directly generate a patch file that resolves the issue.

The second setting, called Shell only, is adapted from the interactive coding framework introduced in Yang et al. (2023a). Following the InterCode environment, this interactive baseline system asks the LM to resolve the issue by interacting with a shell process on Linux. Like SWE-agent, the model prediction is generated automatically based on the final state of the codebase after interaction.

Metrics. We report **% Resolved** as the main metric, which is the proportion of instances for which all tests pass successfully after the model generated patch is applied to the repository (Jimenez et al., 2024). We also report the **$ Avg. Cost** metric, the average API inference cost incurred from running SWE-agent on a task instance, for successful instances. Due to budget constraints we set the per-instance budget to $4. If a run exceeds this budget, the existing edits are submitted automatically.

Configuration Choice. During the design process of SWE-agent, we arrived at the final ACI design through qualitative analysis of system behavior on small set of hand-picked easier examples from the development split of SWE-bench. For the remaining hyperparameter choices, we perform a hyperparameter sweep over the window size, history processing, and decoding temperature. Further description of the configuration choice is provided in Appendix A.

5 Results

Main Results. Across all systems, SWE-agent with GPT-4 Turbo yields the best performance, solving 12.47% (286/2,294) of the full SWE-bench test set and 18.00% (54/300) of the Lite split.

We present several empirical ablations that quantify Agent-Computer Interface design's effect on task performance. We provide insights on language agents' behavior and common failure modes.

5.1 Analysis of Interface Design

Following Section 3, we perform several ablations of the SWE-agent interface, specifically with respect to the SWE-agent w/ GPT-4 set up. The performance of each ablation relative to SWE-agent

[1]Token counts for different models are not directly comparable, since they use different tokenizers.

Table 1: Main results for SWE-agent performance on the full and Lite splits of the SWE-bench test set. We benchmark models in the SWE-agent, Basic CLI, and Retrieval Augmented Generation (RAG) settings established in SWE-bench (Jimenez et al., 2024).

	SWE-bench		SWE-bench Lite	
Model	% Resolved	$ Avg. Cost	% Resolved	$ Avg. Cost
RAG				
w/ GPT-4 Turbo	1.31	0.13	2.67	0.13
w/ Claude 3 Opus	3.79	0.25	4.33	0.25
Shell-only agent				
w/ GPT-4 Turbo	-	-	11.00	1.46
w/o Demonstration	-	-	7.33	0.79
SWE-agent				
w/ GPT-4 Turbo	**12.47**	1.59	**18.00**	1.67
w/ Claude 3 Opus	10.46	2.59	13.00	2.18

w/ GPT-4's performance on SWE-bench Lite is denoted by the red arrow. Our case studies shed light on interesting agent behavior along with the impact of different ACI designs.

Table 2: SWE-bench Lite performance under ablations to the SWE-agent interface, which is denoted by 🤠. We consider different approaches to searching and editing (see Figures 4 and 5 respectively) and study how information should be presented to the language model by varying the window size of the file viewer and comparing to agents without a demonstration or without history management.

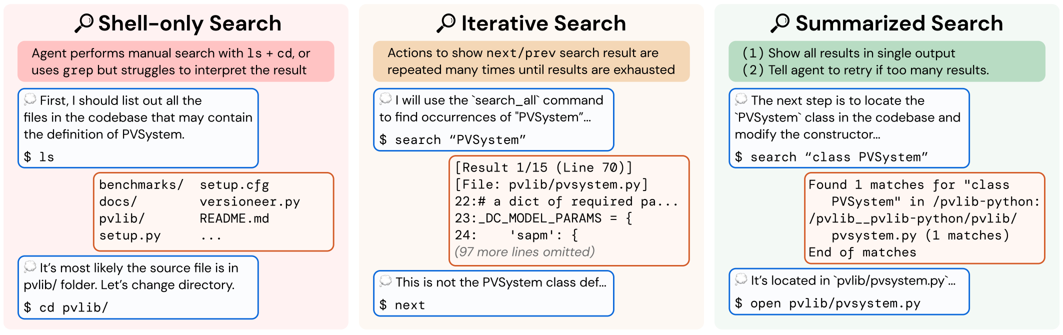
Editor	
`edit` action	15.0 ↓ 3.0
w/ linting 🤠	18.0
Shell-only	10.3 ↓ 7.7

Search	
Summarized 🤠	18.0
Iterative	12.0 ↓ 6.0
Shell-only	15.7 ↓ 2.3

File Viewer	
30 lines	14.3 ↓ 3.7
100 lines 🤠	18.0
400 lines	17.0 ↓ 1.0
Full file	12.7 ↓ 5.3

Context	
w/ demo. 🤠	18.0
Full history	15.0 ↓ 3.0
w/o demo.	16.3 ↓ 1.7

Human user interfaces are not always suitable as agent-computer interfaces. Current LMs are vulnerable to a number of pitfalls when searching for relevant content in the Shell-only environment. Some exploration patterns (e.g. chains of `cd`, `ls`, `cat`) are extremely inefficient. `grep` or `find` look ups can be better, but occasionally produce many lines of irrelevant results. We hypothesize that better localization is possible with faster navigation and a more informative search interface.



Agent performs manual search with `ls` + cd, or uses `grep` but struggles to interpret the result

First, I should list out all the files in the codebase that may contain the definition of PVSystem.

```
$ ls
```

```
benchmarks/   setup.cfg
docs/         versioneer.py
pvlib/        README.md
setup.py      ...
```

It's most likely the source file is in pvlib/ folder. Let's change directory.

```
$ cd pvlib/
```

Actions to show `next/prev` search result are repeated many times until results are exhausted

I will use the `search_all` command to find occurrences of "PVSystem"...

```
$ search "PVSystem"
```

```
[Result 1/15 (Line 70)]
[File: pvlib/pvsystem.py]
22:# a dict of required pa...
23:_DC_MODEL_PARAMS = {
24:    'sapm': {
(97 more lines omitted)
```

This is not the PVSystem class def...

```
$ next
```

(1) Show all results in single output
(2) Tell agent to retry if too many results.

The next step is to locate the `PVSystem` class in the codebase and modify the constructor...

```
$ search "class PVSystem"
```

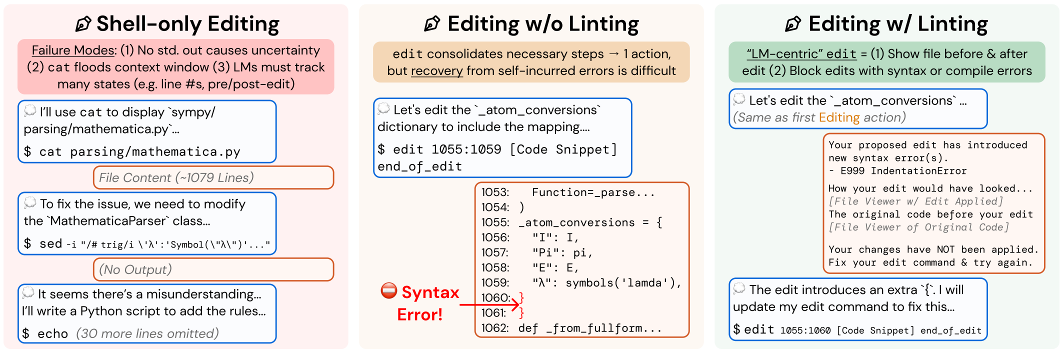
```
Found 1 matches for "class
   PVSystem" in /pvlib-python:
/pvlib__pvlib-python/pvlib/
   pvsystem.py (1 matches)
End of matches
```

It's located in `pvlib/pvsystem.py`...

```
$ open pvlib/pvsystem.py
```

Figure 4: Visualizations of three different versions of the Search interface for task instance `pvlib__pvlib-python-1224`. In Shell-only, an agent performs localization strictly with standard bash commands and utilities. Compared to *Iterative* search, *Summarized* search shows an exhaustive list of search results and also provides guidance on refining underspecified queries.

Figure 4 compares the Shell-only setting with two different search interfaces. *Iterative* search, directly inspired by traditional user interfaces for search, such as `Vim` or VSCode, shows results one by one via the file viewer. Agents can look through results using `next` and `prev` actions. Each result displays the matching line along with `n` surrounding lines of context. An advantage is that an agent can begin editing directly after seeing the relevant code in its search. However, when given a large number of search results, agents tend to look through every match exhaustively, calling `next`, until each result has been inspected. This inefficient behavior can exhaust an agent's cost budget or context window, leading to performance even worse than the Shell-only setting (15.7%$_{\downarrow 2.3}$ for Shell-only vs. 12.0%$_{\downarrow 6.0}$ with Iterative search).



Shell-only Editing

Failure Modes: (1) No std. out causes uncertainty (2) `cat` floods context window (3) LMs must track many states (e.g. line #s, pre/post-edit)

I'll use `cat` to display `sympy/parsing/mathematica.py`...

```
$ cat parsing/mathematica.py
```

File Content (~1079 Lines)

To fix the issue, we need to modify the `MathematicaParser` class...

```
$ sed -i "/# trig/i \'λ':'Symbol(\"λ\")'..."
```

(No Output)

It seems there's a misunderstanding... I'll write a Python script to add the rules...

```
$ echo (30 more lines omitted)
```

Editing w/o Linting

`edit` consolidates necessary steps → 1 action, but recovery from self-incurred errors is difficult

Let's edit the `_atom_conversions` dictionary to include the mapping....

```
$ edit 1055:1059 [Code Snippet]
end_of_edit
```

```
1053:   Function=_parse...
1054: )
1055: _atom_conversions = {
1056:   "I": I,
1057:   "Pi": pi,
1058:   "E": E,
1059:   "λ": symbols('lamda'),
1060: }
1061: }
1062: def _from_fullform...
```

Editing w/ Linting

"LM-centric" `edit` = (1) Show file before & after edit (2) Block edits with syntax or compile errors

Let's edit the `_atom_conversions` ...
(Same as first Editing action)

```
Your proposed edit has introduced
new syntax error(s).
- E999 IndentationError

How your edit would have looked...
[File Viewer w/ Edit Applied]
The original code before your edit
[File Viewer of Original Code]

Your changes have NOT been applied.
Fix your edit command & try again.
```

The edit introduces an extra `{`. I will update my edit command to fix this...

```
$ edit 1055:1060 [Code Snippet] end_of_edit
```

Figure 5: Visualizations of three different versions of the Edit component for task instance `sympy__sympy-24102`. Editing with Bash commands requires several actions to successfully modify a file. The *Editing* component defines an `edit` command that leverages the File Viewer component to replaces the Bash style editing workflow with a single command. *Linting* is beneficial to stymieing cascading errors that often starts with an error-introducing edit by the agent.

Compact, efficient file editing is critical to performance. SWE-agent's file editor and viewer is designed to consolidate the editing process into a single command that allows for easy multi-line edits with consistent feedback that automatically updates the agent's view of the file after editing. In the Shell-only setting, editing options are restrictive and prone to errors. The primary methods available are either replacing entire files through redirection and overwriting or utilizing utilities like `sed` for single-line or search-and-replace edits. Both methods have significant drawbacks. Redirection involves copying and rewriting entire files for even minor changes, which is both inefficient and error-prone. Although `sed` can facilitate specific edits, executing multi-line edits is cumbersome and can lead to unintended consequences that are challenging to detect. Moreover, both strategies lack immediate feedback about file updates, making these silent operations potentially confusing for models to interpret and increasing the risk of errors. Without SWE-agent's file editor interface, performance drops to (10.3% $_{\downarrow 7.7}$). We also find that agents are sensitive to the number of lines the file viewer displays. Both too little content (30 lines, 14.3% $_{\downarrow 3.7}$) or too much (entire file, 12.7% $_{\downarrow 5.3}$) leads to lower performance.

Guardrails can improve error recovery. A prominent failure mode is when models repeatedly `edit` the same code snippet. The usual suspect for this behavior is when an agent introduces a syntax error (e.g., wrong indentation, extra parenthesis) via an errant `edit`. As discussed in Section 3, we add an intervention to the `edit` logic such that a modification only applies if it does not produce major errors. We compare this interface with the Shell-only and no-linting alternative in Figure 5. This intervention improves performance considerably (Without Linting, 15.0% $_{\downarrow 3.0}$).

5.2 Analysis of Agent Behavior

Consistent problem solving patterns emerge when LMs are equipped with a useful, intuitive ACI. In this section, we discuss several model behaviors and problem solving patterns that can be discerned from model performance and each model's corresponding trajectories.

Reproduction and/or localization is the first step. SWE-agent usually starts a problem by attempting to write reproduction code and/or localizing the issue's cause to specific code based on file names and symbols referenced in the issue description. This is reflected in Table 3 and Figure 6, where four of the most frequent patterns involve performing such diagnostics. To reproduce, models will `create` a new file, add reproduction code to it with an `edit`, then run with `python` (#2 in Table 8). Localization happens both in the absence of and after reproduction; using file names and symbols referenced in the issue description, agents will then invoke `search_dir`, `search_file` (#8, 10), and `find_file` (#9) to identify the root cause. In Figure 6, the distributions for the

Table 3: The 10 most frequently occurring patterns in trajectories of resolved task instances by SWE-agent w/ GPT-4. We define a pattern as 3 consecutive actions.

Pattern	Count
`edit` (3x)	308
`create, edit, python`	215
`edit, python, edit`	196
`edit, python, rm`	157
`python, rm, submit`	142
`python, edit, python`	125
`edit, edit, python`	122
`open, search_file, goto`	107
`edit, python, find_file`	98
`search_file, goto, edit`	82

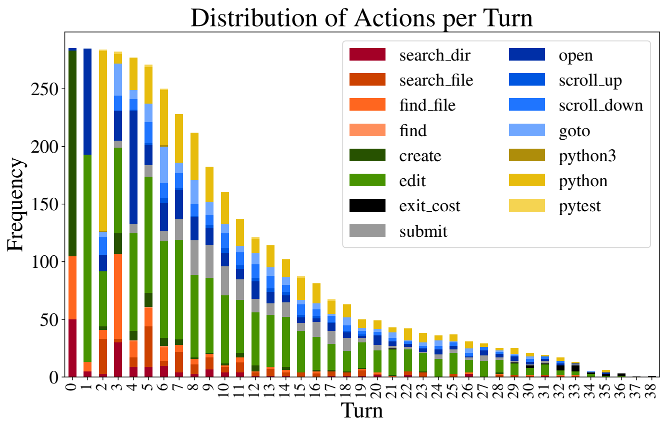


Figure 6: We display the frequency with which actions are invoked at each turn by SWE-agent w/ GPT-4 for task instances that it solved on the SWE-bench full test set (286 trajectories).

first four turns strongly correlate with these trends. Table 8 further corroborates this observation by showing the most frequent patterns per turn.

A majority of turns are spent editing. From turn 5 onwards, the most frequent two actions for all turns are `edit` and `python`. We show a normalized view of Figure 6 in Figure 13 for easy comparison. The dominant pattern, also captured in Tables 3 and 8, is a repetition of editing a file followed by running the original reproduction script to check if the change has the intended effect. For turns 8 to 16, `submit` is the third most invoked action. From manual inspection, our conclusions are that agents rarely deviate from this pattern but they rarely recognize dead ends or reset their problem solving approach.

Agents succeed fast and fail slowly. We find that runs submitted relatively early are much more likely to be successful compared to runs submitted after a larger number of steps or cost. We show in Table 10 the distribution of resolved and unresolved instances, including only instances that did not exhaust their budget. We observe that successful runs complete earlier and at cheaper cost compared unsuccessful submissions. Overall, successful instances solved by SWE-agent w/ GPT 4 finish with an median cost of \$1.21 and 12 steps compared to a mean of \$2.52 and 21 steps for unsuccessful ones. Further, we find that 93.0% of resolved instances are submitted before exhausting their cost budget. For these reasons, we suspect that increasing the maximum budget or token limit are unlikely to lead to substantial increases in performance.

6 Related Work

6.1 Software Engineering Benchmarks

Code generation benchmarks, which evaluate models on the task of synthesizing code from natural language descriptions, have served as a long standing bellwether for measuring LM performance (Chen et al., 2021; Austin et al., 2021; Hendrycks et al., 2021; Lu et al., 2021). Subsequent works have built upon the code generation task formulation to contribute new benchmarks that translate problems to different (programming) languages (Cassano et al., 2022; Wang et al., 2023b), incorporate third party libraries (Lai et al., 2022; Liu et al., 2024b), introduce derivative code completion tasks (Muennighoff et al., 2024; Huang et al., 2024), increase test coverage (Liu et al., 2023), change the edit scope (Ding et al., 2023; Yu et al., 2023; Du et al., 2023), and add robustness to dataset contamination (Jain et al., 2024). Code generation problems are largely self-contained, with short problem descriptions ($\sim$100 lines) and corresponding solutions that are similarly brief, requiring nothing more complex than basic language primitives. Tests are either handwritten or generated synthetically via fuzz testing. In recent months, rapid development of LMs has begun to saturate many of these benchmarks. The top method solves 94.4% of HumanEval (Zhou et al., 2023a).

Gauging future trends with the code generation task paradigm may be limited by the simplicity of this setting and cost of human-in-the-loop problem creation. In response, recent efforts have demonstrated that software engineering (SE) can serve as a diverse, challenging testbed for LM evaluation (Zhang et al., 2023; Jimenez et al., 2024; Liu et al., 2024a). Repository-level code editing introduces many reasoning challenges grounded in real SE subtasks such as spotting errant code, identifying cross-file relationships, and understanding codebase-specific symbols and conventions. As a field, SE has generally studied tasks in a more isolated manner; prior benchmarks tend to frame problems in isolation from the rest of a codebase (Just et al., 2014; Karampatsis & Sutton, 2019).

We use SWE-bench because it unites many separate SE tasks such as automated program repair (Xia & Zhang, 2022; Fan et al., 2023; Sobania et al., 2023), bug localization (Chakraborty et al., 2018; Yang et al., 2024), and testing (Kang et al., 2023; Xia et al., 2023; Wang et al., 2023a) under a single task formulation that faithfully mirrors practical SE. Furthermore, SWE-bench task instances are diverse, having been collected from real GitHub issues across 12 different repositories. In addition, SWE-bench performance is based on rigorous, automatic execution-based evaluation.

6.2 Language Models as Agents

The co-emergence of stronger LMs, increasingly challenging benchmarks, and practical use cases have all together motivated a paradigm shift in LMs' inference setting. In place of traditional zero/few-shot generation, language agents (Sumers et al., 2023; Xi et al., 2023; Wang et al., 2024a), which use LMs to interact with a real/virtual world, have proliferated as the default setting for web navigation (Nakano et al., 2022; Thoppilan et al., 2022; Yao et al., 2023a;b; Zhou et al., 2023b; Sridhar et al., 2023; Press et al., 2023; Koh et al., 2024), computer control (Wu et al., 2024; Xie et al., 2024), and code generation tasks (Yin et al., 2022; Wang et al., 2023c).

Interaction and code generation are increasingly used together, with code as the modality of choice for actions (Yang et al., 2023a; Wang et al., 2024b), tool construction (Wang et al., 2024c; Zhang et al., 2024; Gu et al., 2024), and reasoning (Zelikman et al., 2022; 2024; Shinn et al., 2023). Code language agents have also been applied to offensive security (Yang et al., 2023b; Shao et al., 2024; Fang et al., 2024), theorem proving (Thakur et al., 2024), and clinical tasks (Shi et al., 2024b).

To the best of our knowledge, SWE-agent is the first work to explore language agents for end-to-end software engineering (SE). A key observation we noticed when designing SWE-agent is that many interactive methods, which perform well for rudimentary code completion tasks, do not work for more general settings like SE, a trend also exhibited by baselines for difficult math and competitive programming benchmarks (Shi et al., 2024a).

SWE-agent's takeaways are a reliable indicator of the robustness and scalability of agents' ability to reason with code. In addition, prior works tend to independently explore the merits of tool use, prompting techniques, and code execution in interactive settings. Agent-Computer Interface design accounts for the sum of these factors under a unified framework and introduces the idea that crafting LM-centric interactive components has meaningful effects on downstream task performance.

7 Discussion

We introduce SWE-agent, a language agent for software engineering that achieves state-of-the-art performance on SWE-bench. Through our design methodology, empirical results, and analysis, we demonstrate the process and value of designing agent-computer interfaces (ACI) tailored for agents. We release our code, prompts and generations, and set up the codebase to allow for easy extension to new commands, and feedback and agent history formats. We hope that SWE-agent will serve as a foundation that inspires future work towards more versatile and powerful agents.

Beyond various empirical applications, we hope the further study of ACIs could also make principled use and contribute to our understanding of language models and agents, analogous to the synergy between human-computer interaction (HCI) and psychology (Carroll, 1997). Humans and LMs have different characteristics, training objectives, specialities, and limitations (Griffiths, 2020; McCoy et al., 2023), and the interface design processes can be seen as systematic behavioral experimentation that could reveal more insights into these differences, and establish a comparative understanding of human and artificial intelligence.

Acknowledgements

We thank Austin W. Hanjie, Sam Ainsworth, Xindi Wu, Yuhan Liu, Mengzhou Xia, Dan Friedman, Tianyu Gao, Adithya Bhaskar, Aatmik Gupta, Louisa Nyhus, and Richard Zhu for their valuable feedback and advice. We would also like to thank the broader Princeton Language and Intelligence community for supporting our work. We acknowledge support from an Oracle Collaborative Research award and the National Science Foundation under Grant No. 2239363. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

References

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.

John M Carroll. Human-computer interaction: psychology as a science of design. *Annual review of psychology*, 48(1):61–83, 1997.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, 2022.

Saikat Chakraborty, Yujian Li, Matt Irvine, Ripon Saha, and Baishakhi Ray. Entropy guided spectrum based bug localization using statistical language model. *arXiv preprint arXiv:1802.06947*, 2018.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, and Jared Kaplan et. al. Evaluating large language models trained on code, 2021.

Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024.

Yangruibo Ding, Zijian Wang, Wasi Uddin Ahmad, Hantian Ding, Ming Tan, Nihal Jain, Murali Krishna Ramanathan, Ramesh Nallapati, Parminder Bhatia, Dan Roth, and Bing Xiang. Crosscodeeval: A diverse and multilingual benchmark for cross-file code completion. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=wgDcbBMSfh.

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, 2023.

Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. Automated repair of programs from large language models, 2023.

Richard Fang, Rohan Bindu, Akul Gupta, Qiusi Zhan, and Daniel Kang. Llm agents can autonomously hack websites, 2024.

Thomas L Griffiths. Understanding human intelligence through human limitations. *Trends in Cognitive Sciences*, 24(11):873–883, 2020.

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, 2024.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring coding challenge competence with apps, 2021.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation, 2024.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 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*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of existing faults to enable controlled testing studies for Java programs. In *ISSTA 2014, Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pp. 437–440, San Jose, CA, USA, July 2014. Tool demo.

Sungmin Kang, Juyeon Yoon, and Shin Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction, 2023.

Rafael-Michael Karampatsis and Charles Sutton. How often do single-statement bugs occur? the manysstubs4j dataset. *2020 IEEE/ACM 17th International Conference on Mining Software Repositories (MSR)*, pp. 573–577, 2019. URL https://api.semanticscholar.org/CorpusID:173188438.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks, 2024.

Yuhang Lai, Chengxi Li, Yiming Wang, Tianyi Zhang, Ruiqi Zhong, Luke Zettlemoyer, Scott Wen tau Yih, Daniel Fried, Sida Wang, and Tao Yu. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

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. *arXiv preprint arXiv:2305.01210*, 2023.

Tianyang Liu, Canwen Xu, and Julian McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024a. URL https://openreview.net/forum?id=pPjZIOuQuF.

Yuliang Liu, Xiangru Tang, Zefan Cai, Junjie Lu, Yichi Zhang, Yanjun Shao, Zexuan Deng, Helan Hu, Kaikai An, Ruijun Huang, Shuzheng Si, Sheng Chen, Haozhe Zhao, Liang Chen, Yan Wang, Tianyu Liu, Zhiwei Jiang, Baobao Chang, Yujia Qin, Wangchunshu Zhou, Yilun Zhao, Arman Cohan, and Mark Gerstein. Ml-bench: Evaluating large language models for code generation in repository-level machine learning tasks, 2024b.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

R Thomas McCoy, Shunyu Yao, Dan Friedman, Matthew Hardy, and Thomas L Griffiths. Embers of autoregression: Understanding large language models through the problem they are trained to solve. *arXiv preprint arXiv:2309.13638*, 2023.

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*, 2024. URL https://openreview.net/forum?id=mw1PWNSWZP.

Reiichiro Nakano, Jacob Hilton, Suchir Balaji, Jeff Wu, Long Ouyang, Christina Kim, Christopher Hesse, Shantanu Jain, Vineet Kosaraju, William Saunders, Xu Jiang, Karl Cobbe, Tyna Eloundou, Gretchen Krueger, Kevin Button, Matthew Knight, Benjamin Chess, and John Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. Gpt-4 technical report, 2023.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. Measuring and narrowing the compositionality gap in language models. In Houda Bouamor, Juan

Pino, and Kalika Bali (eds.), *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 5687–5711, Singapore, December 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378.

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, 2024.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. Can language models solve olympiad programming?, 2024a.

Wenqi Shi, Ran Xu, Yuchen Zhuang, Yue Yu, Jieyu Zhang, Hang Wu, Yuanda Zhu, Joyce Ho, Carl Yang, and May D. Wang. Ehragent: Code empowers large language models for few-shot complex tabular reasoning on electronic health records, 2024b.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

Dominik Sobania, Martin Briesch, Carol Hanna, and Justyna Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023.

Abishek Sridhar, Robert Lo, Frank F. Xu, Hao Zhu, and Shuyan Zhou. Hierarchical prompting assists large language model on web navigation, 2023.

Theodore Sumers, Shunyu Yao, Karthik Narasimhan, and Thomas L. Griffiths. Cognitive architectures for language agents, 2023.

Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving, 2024.

Romal Thoppilan, Daniel De Freitas, Jamie Hall, Noam Shazeer, Apoorv Kulshreshtha, Heng-Tze Cheng, Alicia Jin, Taylor Bos, Leslie Baker, Yu Du, YaGuang Li, Hongrae Lee, Huaixiu Steven Zheng, Amin Ghafouri, Marcelo Menegali, Yanping Huang, Maxim Krikun, Dmitry Lepikhin, James Qin, Dehao Chen, Yuanzhong Xu, Zhifeng Chen, Adam Roberts, Maarten Bosma, Vincent Zhao, Yanqi Zhou, Chung-Ching Chang, Igor Krivokon, Will Rusch, Marc Pickett, Pranesh Srinivasan, Laichee Man, Kathleen Meier-Hellstern, Meredith Ringel Morris, Tulsee Doshi, Renelito Delos Santos, Toju Duke, Johnny Soraker, Ben Zevenbergen, Vinodkumar Prabhakaran, Mark Diaz, Ben Hutchinson, Kristen Olson, Alejandra Molina, Erin Hoffman-John, Josh Lee, Lora Aroyo, Ravi Rajakumar, Alena Butryna, Matthew Lamm, Viktoriya Kuzmina, Joe Fenton, Aaron Cohen, Rachel Bernstein, Ray Kurzweil, Blaise Aguera-Arcas, Claire Cui, Marian Croak, Ed Chi, and Quoc Le. Lamda: Language models for dialog applications, 2022.

Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language model: Survey, landscape, and vision, 2023a.

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, Wayne Xin Zhao, Zhewei Wei, and Jirong Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), March 2024a. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024b.

Zhiruo Wang, Grace Cuenca, Shuyan Zhou, Frank F. Xu, and Graham Neubig. Mconala: A benchmark for code generation from multiple natural languages, 2023b.

Zhiruo Wang, Shuyan Zhou, Daniel Fried, and Graham Neubig. Execution-based evaluation for open-domain code generation, 2023c.

Zhiruo Wang, Daniel Fried, and Graham Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks, 2024c.

Zhiyong Wu, Chengcheng Han, Zichen Ding, Zhenmin Weng, Zhoumianze Liu, Shunyu Yao, Tao Yu, and Lingpeng Kong. Os-copilot: Towards generalist computer agents with self-improvement, 2024.

Zhiheng Xi, Wenxiang Chen, Xin Guo, Wei He, Yiwen Ding, Boyang Hong, Ming Zhang, Junzhe Wang, Senjie Jin, Enyu Zhou, Rui Zheng, Xiaoran Fan, Xiao Wang, Limao Xiong, Yuhao Zhou, Weiran Wang, Changhao Jiang, Yicheng Zou, Xiangyang Liu, Zhangyue Yin, Shihan Dou, Rongxiang Weng, Wensen Cheng, Qi Zhang, Wenjuan Qin, Yongyan Zheng, Xipeng Qiu, Xuanjing Huang, and Tao Gui. The rise and potential of large language model based agents: A survey, 2023.

Chunqiu Steven Xia and Lingming Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 959–971, 2022.

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. Universal fuzzing via large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2023.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024.

Aidan Z. H. Yang, Claire Le Goues, Ruben Martins, and Vincent Hellendoorn. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400702174. doi: 10.1145/3597503.3623342. URL https://doi.org/10.1145/3597503.3623342.

John Yang, Akshara Prabhakar, Karthik R Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023a. URL https://openreview.net/forum?id=fvKaLF1ns8.

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*, 2023b.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023b. URL https://openreview.net/forum?id=WE_vluYUL-X.

Pengcheng Yin, Wen-Ding Li, Kefan Xiao, Abhishek Rao, Yeming Wen, Kensen Shi, Joshua Howland, Paige Bailey, Michele Catasta, Henryk Michalewski, Alex Polozov, and Charles Sutton. Natural language to code generation in interactive data science notebooks, 2022.

Hao Yu, Bo Shen, Dezhi Ran, Jiaxin Zhang, Qi Zhang, Yu Ma, Guangtai Liang, Ying Li, Tao Xie, and Qianxiang Wang. Codereval: A benchmark of pragmatic code generation with generative pre-trained models. In *International Conference on Software Engineering*, 2023. URL https://api.semanticscholar.org/CorpusID:256459413.

Eric Zelikman, Qian Huang, Gabriel Poesia, Noah D Goodman, and Nick Haber. Parsel: Algorithmic reasoning with language models by composing decompositions, 2022. URL https://arxiv.org/abs/2212.10561.

Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (stop): Recursively self-improving code generation, 2024.

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. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://openreview.net/forum?id=q09vTY1Cqh.

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

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning acting and planning in language models, 2023a.

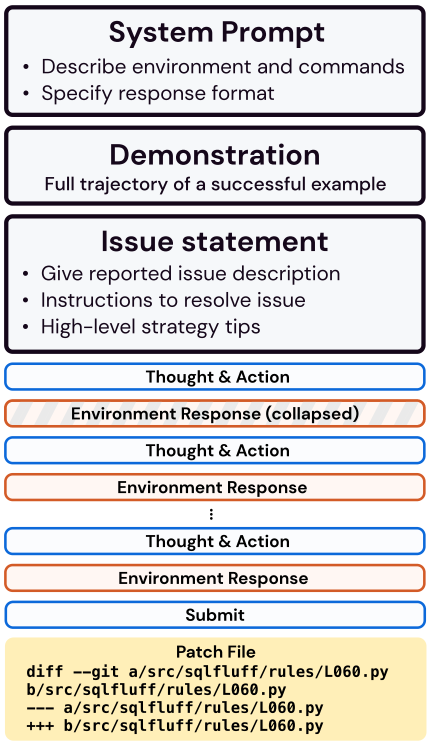
Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2023b.

Appendix

In the appendix, we provide additional analyses and more extensive discussions about SWE-agent, Agent Computer Interface (ACI) Design, and model performance on the full and Lite splits of SWE-bench. We also provide several thorough, manually curated case studies of SWE-agent performance on select task instances.

A SWE-agent Interface

In this section, we go into greater discussion about the design methodology, appearance, and implementation of each of the SWE-agent components. As described in Section 3, the SWE-agent interface is consists of several components that enable agents to accomplish key sub-tasks that are fundamental to solving software engineering problems. Namely: localizing which file(s) require editing, generating edits to fix the described issue, and writing scripts to verify the correctness of fixes. To enable LM-based agents to efficiently carry out these individual functions and progress towards the overarching goal of resolving a codebase issue, we provide a file viewer, file editor, search / navigation system, and context management system. In Section A.1, we provide a thorough breakdown of each of these components. In Section A.2, we discuss how SWE-agent is configured to support the final interface, along with how SWE-agent is built to enable easy extensibility and customization to alter the interface. Finally, in Section A.3, we discuss the technical design decisions and challenges of building out SWE-agent, along with an overview of the advantages and shortcomings of using SWE-agent as a platform for future explorations of agent-driven software engineering systems.


System Prompt
• Describe environment and commands
• Specify response format
Demonstration
Full trajectory of a successful example
Issue statement
• Give reported issue description
• Instructions to resolve issue
• High-level strategy tips
Thought & Action
Environment Response (collapsed)
Thought & Action
Environment Response
⋮
Thought & Action
Environment Response
Submit
Patch File

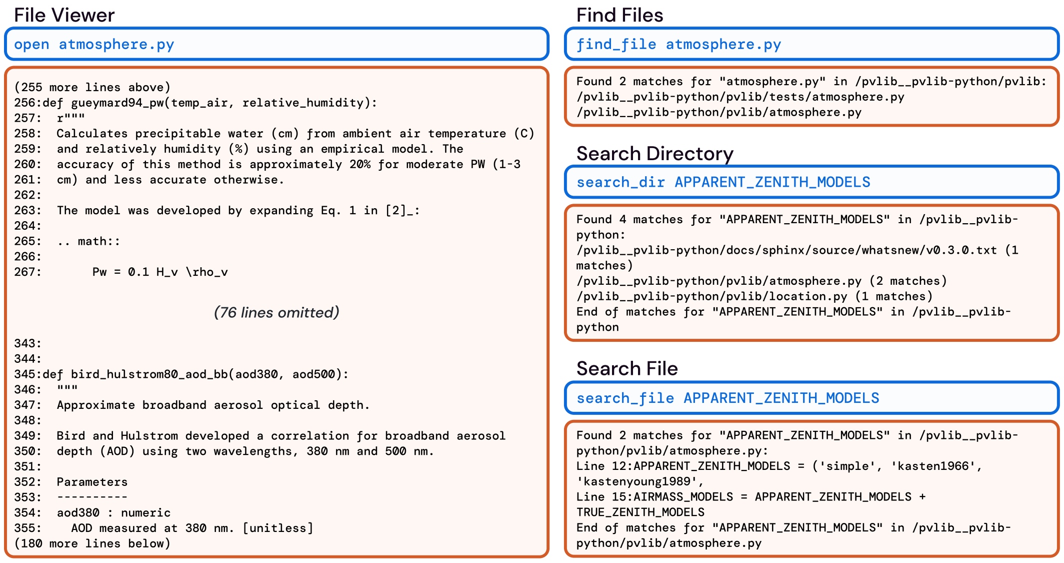
```
diff --git a/src/sqlfluff/rules/L060.py
b/src/sqlfluff/rules/L060.py
--- a/src/sqlfluff/rules/L060.py
+++ b/src/sqlfluff/rules/L060.py
```

Figure 7: An overview over the structure of a trajectory: We first present the system prompt, demonstration (optional), and issue statement. The agent then interacts in turn with the environment. Past observations may be *collapsed*, i.e. we truncate any long output, as described in Section 3.

A.1 Component Design

In this section, we revisit each component discussed in Section 3. First, we describe the motivation for the component, then provide complete descriptions of their input requirement(s), output format, usage, and notes about what parts of the interface heavily influence language model behavior.

File Viewer. As discussed in Section 3, the File Viewer is fundamental to a language agent's ability to understand file content and invoke appropriate edits. In a Terminal-only setting, there are several commands that can be used to inspect file content. However, out of the box command line tools are sub-optimal or limiting for language agents for several reasons. First, commands that print files to standard output (e.g. `cat`, `printf`) can easily flood a language agent's context window with too much file content, the majority of which is usually irrelevant to the issue. Enabling a language agent to filter out distractions and focus on relevant code snippets is crucial to generating effective edits. While commands like `head` and `tail` reduce length to the first/last `n` lines, it is not intuitive to use bash commands to perform in-file navigation. It is either impossible or requires a long list of arguments to show specific file lines. Furthermore, since such Bash commands are stateless, "scrolling" up/down relative to the current file position typically requires regenerating the same lengthy command with minor changes. Interactive tools like `more` and `less` accommodate this, but (1) representing navigation actions (multiple key up/down clicks) is intuitive for humans, but is verbose and costly for language agents, and (2) even if jumping to a



File Viewer

```
open atmosphere.py
```

```
(255 more lines above)
256:def gueymard94_pw(temp_air, relative_humidity):
257:  r"""
258:  Calculates precipitable water (cm) from ambient air temperature (C)
259:  and relatively humidity (%) using an empirical model. The
260:  accuracy of this method is approximately 20% for moderate PW (1-3
261:  cm) and less accurate otherwise.
262:
263:  The model was developed by expanding Eq. 1 in [2]_:
264:
265:  .. math::
266:
267:       Pw = 0.1 H_v \rho_v

                         (76 lines omitted)

343:
344:
345:def bird_hulstrom80_aod_bb(aod380, aod500):
346:  """
347:  Approximate broadband aerosol optical depth.
348:
349:  Bird and Hulstrom developed a correlation for broadband aerosol
350:  depth (AOD) using two wavelengths, 380 nm and 500 nm.
351:
352:  Parameters
353:  ----------
354:  aod380 : numeric
355:    AOD measured at 380 nm. [unitless]
(180 more lines below)
```

Find Files

```
find_file atmosphere.py
```

```
Found 2 matches for "atmosphere.py" in /pvlib__pvlib-python/pvlib:
/pvlib__pvlib-python/pvlib/tests/atmosphere.py
/pvlib__pvlib-python/pvlib/atmosphere.py
```

Search Directory

```
search_dir APPARENT_ZENITH_MODELS
```

```
Found 4 matches for "APPARENT_ZENITH_MODELS" in /pvlib__pvlib-
python:
/pvlib__pvlib-python/docs/sphinx/source/whatsnew/v0.3.0.txt (1
matches)
/pvlib__pvlib-python/pvlib/atmosphere.py (2 matches)
/pvlib__pvlib-python/pvlib/location.py (1 matches)
End of matches for "APPARENT_ZENITH_MODELS" in /pvlib__pvlib-
python
```

Search File

```
search_file APPARENT_ZENITH_MODELS
```

```
Found 2 matches for "APPARENT_ZENITH_MODELS" in /pvlib__pvlib-
python/pvlib/atmosphere.py:
Line 12:APPARENT_ZENITH_MODELS = ('simple', 'kasten1966',
'kastenyoung1989',
Line 15:AIRMASS_MODELS = APPARENT_ZENITH_MODELS +
TRUE_ZENITH_MODELS
End of matches for "APPARENT_ZENITH_MODELS" in /pvlib__pvlib-
python/pvlib/atmosphere.py
```

Figure 8: The File Viewer and Search components of the SWE-agent interface. The corresponding commands for each component are shown in blue. These examples are copied from trajectories generated by SWE-agent w/ GPT-4 Turbo on the `pvlib__pvlib-python-1603` task instance.

specific line number is allowed, it is not possible to quickly identify what classes/methods/symbols are declared in a file and go to their definitions.

We provide a visualization of the File Viewer component in Figure 8.

A.2 Configuration

The SWE-agent system is instantiated by three functional components: a language model, a configuration, and a shell process running in a Docker container.

An agent-computer interface is made up of four categories of configurable components:

1. Prompt templates - templates prompts are used to inform the language model of the environment and API, augment environment responses with the values of state variables, and provide the initial task setting.
2. Command files - these files contain the source code of bash or Python functions and scripts. Commands are easily modified, added, and removed through manipulating these files' code contents directly. Documentation added in these files can also be injected into prompts to inform the model of the available commands.
3. Control flow - methods for parsing model responses and processing history can be specified through these configuration arguments.
4. Environment variables - initial values of variables that may interact with commands and the shell can also be specified in the configuration.

A.3 Implementation

Table 4: In additional to the standard Linux Bash commands, we provide SWE-agent with specialized tools, including an interactive file viewer, search functionalities, and edit tools for the open file. Required arguments are enclosed in <> and optional arguments are in `[]`.

Category	Command	Documentation
File viewer	**open** `<path> [<line_number>]`	Opens the file at the given path in the editor. If `line_number` is provided, the window will move to include that line.
	goto `<line_number>`	Moves the window to show line_number.
	scroll_down	Moves the window up 100 lines.
	scroll_up	Moves the window down 100 lines.
Search tools	**search_file** `<search_term> [<file>]`	Searches for `search_term` in file. If file is not provided, searches in the current open file.
	search_dir `<search_term> [<dir>]`	Searches for `search_term` in all files in dir. If dir is not provided, searches in the current directory.
	find_file `<file_name> [<dir>]`	Finds all files with the given name in dir. If dir is not provided, searches in the current directory.
File editing	**edit** `<n>`**:**`<m>` `<replacement_text>` **end_of_edit**	Replaces lines n through m (inclusive) with the given text in the open file. All of the `replacement_text` will be entered, so make sure your indentation is formatted properly. Python files will be checked for syntax errors after the edit. If an error is found, the edit will not be executed. Reading the error message and modifying your command is recommended as issuing the same command will return the same error.
	create `<filename>`	Creates and opens a new file with the given name.
Task	**submit**	Generates and submits the patch from all previous edits and closes the shell.

Table 5: Hyper parameter sweep results on a subset of the `dev` split. % Resolved shows the mean score across 5 samples.

Model	Temperature	Window	History	% Resolved
GPT-4 Turbo	0.0	100	Full	14.1
GPT-4 Turbo	0.0	100	Last 5	**15.1**
GPT-4 Turbo	0.0	200	Full	9.2
GPT-4 Turbo	0.0	200	Last 5	10.8
GPT-4 Turbo	0.2	100	Full	10.8
GPT-4 Turbo	0.2	100	Last 5	12.4
GPT-4 Turbo	0.2	200	Full	8.7
GPT-4 Turbo	0.2	200	Last 5	10.8
Claude 3 Opus	0.0	100	Full	5.4
Claude 3 Opus	0.0	100	Last 5	**8.1**
Claude 3 Opus	0.0	200	Full	7.0
Claude 3 Opus	0.0	200	Last 5	7.1
Claude 3 Opus	0.2	100	Full	7.4
Claude 3 Opus	0.2	100	Last 5	**8.1**
Claude 3 Opus	0.2	200	Full	**8.1**
Claude 3 Opus	0.2	200	Last 5	6.8

A.4 Hyperparameter Sweep

We performed a hyperparameter sweep using a subset of 37 instances sampled randomly from the `dev` split of SWE-bench.

B Extended Results

In this section, we provide additional results, including performance marginalized against different dimensions, patch generation statistics, and problem solving patterns reflected by SWE-agent trajectories. Per analysis, we provide numerical or qualitative evidence that supports our findings, describe our takeaways from each finding, and discuss both the strengths of SWE-agent relative to prior baselines along with future directions based on improving common failure modes in SWE-agent's performance.

B.1 Model Performance

We present analyses of model performance marginalized across different dimensions and categories.

Performance by Repository. We include a repository-by-repository breakdown of model performance on the SWE-bench Lite dataset in Table 6. We also include and adjust the performance of Claude 2 on SWE-bench, inherited from the baseline performances established in the original work. As presented above, SWE-agent performance is superior to prior approaches, solving not only a higher percentage of problems across repositories, but also resolving problems in repositories that were previously nearly or completely unsolved by prior approaches (e.g. `matplotlib/matplotlib`, `sympy/sympy`).

Temporal Analysis. In Table 7, we provide a temporal breakdown that shows the % Resolved statistics for task instances from different years. There is no clear correlation between a task instance's creation year and its resolution rate across either models or setting. For instance, while the SWE-agent w/ GPT-4 approach solves the highest percentage of problems from 2021, the RAG w/ GPT-4 and SWE-agent w/ Claude 3 Opus approaches perform best on task instances from 2022.

Table 6: % Resolved performance across repositories represented in the SWE-bench Lite dataset. Each row corresponds to a repository while each column is the model's performance for that repository. The numbers in parentheses in the "Repo" column is the number of task instances in SWE-bench Lite that are from the corresponding repository.

	SWE-agent		RAG		
Repo	GPT 4	Claude 3 Opus	GPT 4	Claude 3 Opus	Claude 2
astropy/astropy (6)	16.67%	33.33%	0.00%	0.00%	0.00%
django/django (114)	26.32%	16.67%	4.39%	6.14%	5.26%
matplotlib/matplotlib (23)	13.04%	13.04%	0.00%	0.00%	0.00%
mwaskom/seaborn (4)	25.00%	0.00%	25.00%	25.00%	0.00%
pallets/flask (3)	0.00%	0.00%	0.00%	0.00%	0.00%
psf/requests (6)	33.33%	16.67%	0.00%	0.00%	0.00%
pydata/xarray (5)	0.00%	0.00%	20.00%	20.00%	0.00%
pylint-dev/pylint (6)	16.67%	0.00%	0.00%	0.00%	0.00%
pytest-dev/pytest (17)	17.65%	5.88%	0.00%	5.88%	5.88%
scikit-learn/scikit-learn (23)	17.39%	17.39%	0.00%	4.35%	8.70%
sphinx-doc/sphinx (16)	6.25%	6.25%	0.00%	0.00%	0.00%
sympy/sympy (77)	10.39%	5.19%	1.30%	2.60%	0.00%

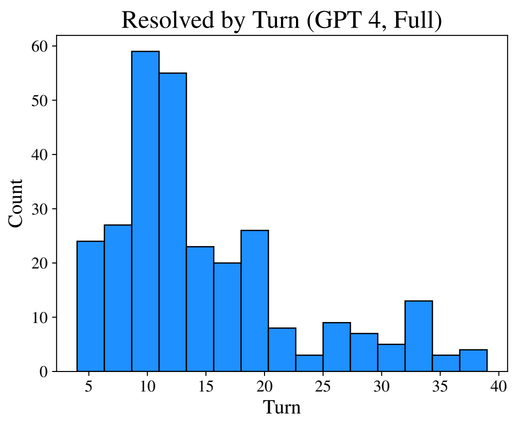
Table 7: % Resolved performance for task instances from different years represented in the SWE-bench Lite dataset. Each row corresponds to a year while each column is the model's performance for task instances with a `created_at` timestamp from that year. The numbers in parentheses in the Year column is the number of task instances in SWE-bench Lite from that corresponding year.

	SWE-agent		RAG		
Year	GPT 4	Claude 3 Opus	GPT 4	Claude 3 Opus	Claude 2
2023 (30)	23.33%	13.33%	3.33%	3.33%	0.0%
2022 (57)	21.05%	17.54%	5.26%	7.02%	1.75%
2021 (42)	23.81%	11.9%	2.38%	4.76%	2.38%
2020 (66)	10.61%	7.58%	3.03%	1.52%	1.52%
Before 2020 (105)	17.14%	10.48%	0.95%	4.76%	5.71%

B.2 Trajectory Analysis

Turns to Resolution. Figure 9 visualizes the distribution of the number of turns SWE-agent needed to complete task instances that were successfully resolved. On the full SWE-bench test set, SWE-agent w/ GPT-4 takes an average of 14.71 turns to finish a trajectory, with a median of 12 turns and 75% of trajectories being completed within 18 turns. On the Lite split of the SWE-bench test set, SWE-agent w/ Claude 3 Opus takes an average of 12.71 turns to finish a trajectory, with a median of 13 turns and 75% of trajectories being completed within 15 turns. From the distribution, it is evident that across models and SWE-bench splits, the majority of task instances are typically solved and completed comfortably within the number of allotted turns. This also points to a general area of improvement for language agent systems - if a language agent's initial problem solving approach, typically reflected in the first 10 to 20 turns, does not yield a good solution, it struggles to make use of later turns that build upon past mistakes. To remedy this issue and induce stronger error recovery capabilities in language agents, future directions could consider improving either the model, the companion ACI, or both. While modeling innovations and better data for language models is a heavily investigated ongoing research direction, we purport that building more adaptive and powerful ACIs may be a much more efficient and generalizable solution for solving difficult interactive tasks with language agents. For instance, one can imagine improving the context management system such that it can recognize unsuccessful or unproductive turns in a problem solving trajectory, then synthesize these turns into a more succinct and helpful error message that prompts the model to attempt a different problem solving approach which avoids repeating past faults.

Figure 9: Distribution of the number of turns for interactive trajectories corresponding to solved task instances. The left histogram shows this distribution for SWE-agent w/ GPT 4 on the full SWE-bench test set (286 trajectories). The right histogram is the performance of SWE-agent w/ Claude 3 Opus on the Lite split of the SWE-bench test set (35 trajectories).



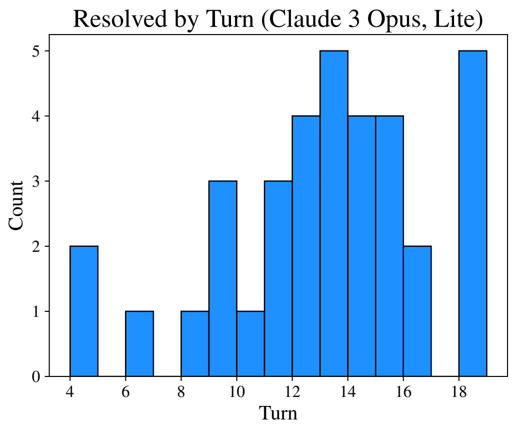
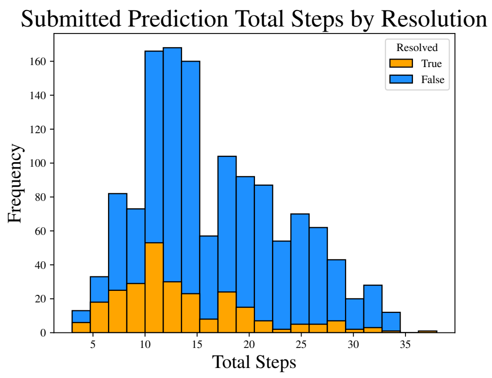
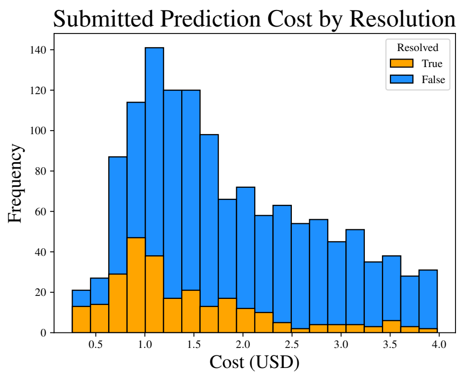


Figure 10: The distribution of agent trajectories by total steps (left) and cost (right) for SWE-agent with GPT-4 Turbo on SWE-bench. The distributions of resolved instances are shown in orange and unresolved are shown in blue. Resolved instances clearly display an earlier mean and fewer proportion of trajectories with many steps or that cost near the maximum budget of $4.00.





B.3 Performance Variance and Pass@$k$ate

Since running SWE-agent on SWE-bench can be rather expensive, we perform, all results, unless otherwise stated, are reported using a pass@1 metric (% Resolved). However, we also test our main SWE-agent configuration for a higher number of runs to test the variance and pass@k performance for $k \in \{3, 6\}$. These results are shown in Table 9, suggesting that average performance variance is relatively low, though per-instance resolution can change considerably.

Figure 11: We assign each pattern to one of five categories (as presented in Table 8) and present a histogram of the turns at which patterns from specific categories show up frequently.

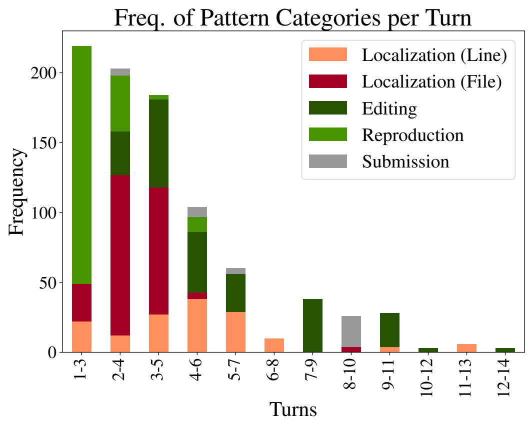


Figure 12: This density plot shows a normalized distribution of actions across different turns of a trajectory. `exit_cost` refers to when the token budget cost was exhausted and the episode's changes are automatically submitted (contrary to an intentional `submit` invoked by the agent).

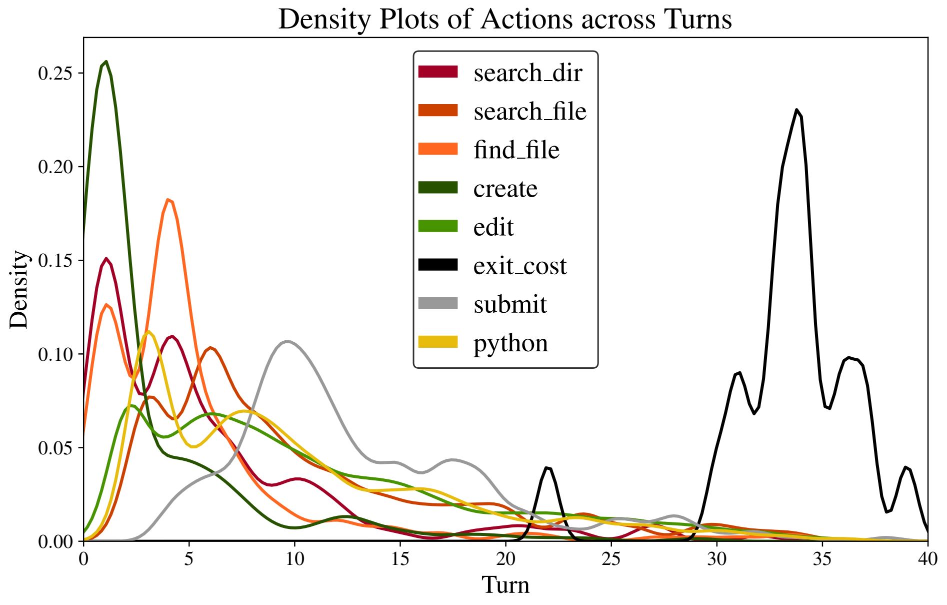


Table 8: Similar to Table 3, we present a table of the most frequently occurring action patterns at each turn (“frequently” means $\geq$ 4 times) in trajectories of task instances resolved by SWE-agent w/ GPT-4. For instance, the pattern `create,edit,python` appears 156 times at the first to third turns. In addition, we also manually assign each entry a category (Reproduction, Editing, Localization (File), Localization (Line), Submission) that generally captures the underlying purpose of such a pattern.

Turns	Pattern	Count	Category
1-3	`create`, `edit`, `python`	156	Reproduction
1-3	`search_dir`, `open`, `search_file`	21	Localization (File)
1-3	`search_dir`, `open`, `scroll_down`	12	Localization (Line)
1-3	`create`, `edit`, `edit`	11	Reproduction
1-3	`search_dir`, `open`, `edit`	10	Localization (Line)
2-4	`edit`, `python`, `find_file`	71	Localization (File)
2-4	`edit`, `python`, `edit`	37	Reproduction
2-4	`edit`, `python`, `search_dir`	26	Localization (File)
2-4	`edit`, `python`, `open`	15	Localization (File)
2-4	`open`, `edit`, `edit`	13	Editing
2-4	`open`, `edit`, `create`	13	Editing
2-4	`open`, `scroll_down`, `scroll_down`	9	Localization (Line)
2-4	`open`, `scroll_down`, `edit`	5	Editing
2-4	`open`, `edit`, `submit`	5	Submission
3-5	`python`, `find_file`, `open`	61	Localization (File)
3-5	`python`, `edit`, `python`	25	Editing
3-5	`search_file`, `goto`, `edit`	24	Localization (Line)
3-5	`python`, `search_dir`, `open`	23	Localization (File)
3-5	`edit`, `create`, `edit`	13	Editing
3-5	`python`, `edit`, `edit`	11	Editing
3-5	`python`, `open`, `edit`	7	Editing
3-5	`python`, `find_file`, `find_file`	7	Localization (File)
3-5	`edit`, `edit`, `submit`	4	Submission
3-5	`edit`, `edit`, `create`	4	Editing
4-6	`find_file`, `open`, `edit`	28	Editing
4-6	`find_file`, `open`, `search_file`	19	Localization (Line)
4-6	`edit`, `edit`, `python`	11	Reproduction
4-6	`goto`, `edit`, `edit`	8	Editing
4-6	`find_file`, `open`, `goto`	8	Localization (Line)
4-6	`goto`, `edit`, `submit`	7	Submission
4-6	`goto`, `edit`, `create`	7	Editing
4-6	`find_file`, `open`, `scroll_down`	6	Localization (Line)
4-6	`scroll_down`, `scroll_down`, `edit`	5	Localization (Line)
4-6	`find_file`, `find_file`, `open`	5	Localization (File)
5-7	`open`, `search_file`, `goto`	29	Localization (Line)
5-7	`open`, `edit`, `python`	20	Editing
5-7	`open`, `goto`, `edit`	7	Editing
5-7	`scroll_down`, `edit`, `submit`	4	Submission
6-8	`scroll_down` (x3)	6	Localization (Line)
6-8	`search_file`, `goto`, `scroll_down`	4	Localization (Line)
7-9	`edit`, `python`, `rm`	20	Editing
7-9	`goto`, `edit`, `python`	12	Editing
8-10	`python`, `rm`, `submit`	19	Submission
8-10	`search_file`, `goto`, `search_file`	4	Localization (File)
9-11	`edit` (x3)	18	Editing
9-11	`edit`, `open`, `edit`	6	Editing
9-11	`goto`, `search_file`, `goto`	4	Localization (Line)

Figure 13: A normalized view of Table 6. The distributions for turn `n` are normalized across the number of trajectories that have a length of at least `n` or more turns.

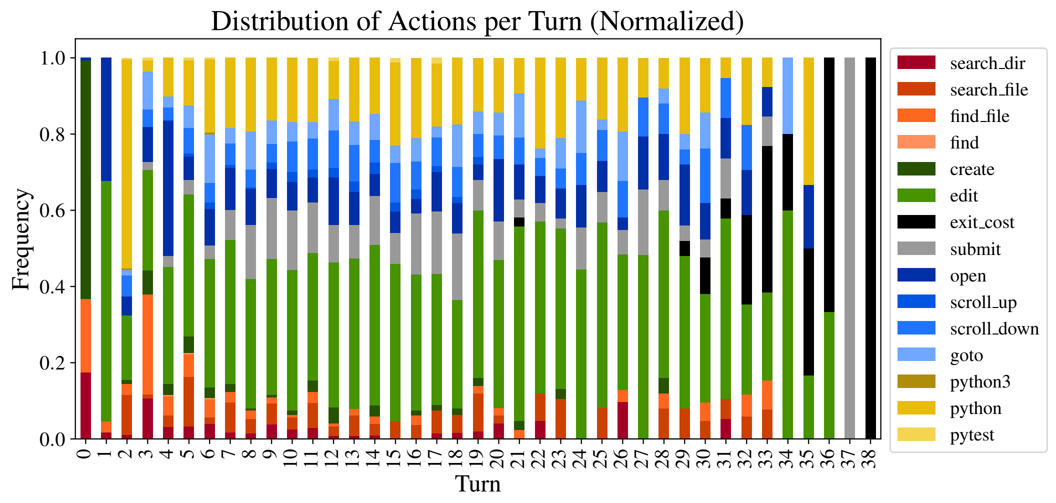


Table 9: Pass@k performance for SWE-agent with GPT-4 Turbo on SWE-bench Lite.

	SWE-bench Lite		
Model	pass@1	pass@3	pass@6
SWE-agent w/ GPT-4 Turbo	17.94%	27.35%	32.67%

C Prompts

We provide a visualization of the variety of prompts used as part of SWE-agent.

Figure 14: The environment's response template. This is emitted after each observation to inform the model of the current state of the shell and programs.

Response Template

```
{OBSERVATION}
(Open file: /path/to/open/file.py)
(Current directory: /path/to/cwd)
bash-$
```

Figure 15: The environment's error message. This is emitted if a model generation doesn't conform to the thought-action format suggested.

Error Message

~~~
Your output was not formatted correctly. You must always include one
discussion and one command as part of your response. Make sure you do
not have multiple discussion/command tags.
Please make sure your output precisely matches the following format:
DISCUSSION
Discuss here with yourself about what your planning and what you're
going to do in this step.

```
command(s) that you're going to run
```
~~~

Figure 16: The system prompt for SWE-agent describes the environment. The `documentation` field is populated with brief description of all enabled commands, similar to Table 4.

System Prompt

SETTING: You are an autonomous programmer, and you're working directly in the command line with a special interface.

The special interface consists of a file editor that shows you 100 lines of a file at a time. In addition to typical bash commands, you can also use the following commands to help you navigate and edit files.

COMMANDS: {documentation}

Please note that THE EDIT COMMAND REQUIRES PROPER INDENTATION. If you'd like to add the line '   print(x)' you must fully write that out, with all those spaces before the code! Indentation is important and code that is not indented correctly will fail and require fixing before it can be run.

RESPONSE FORMAT:
Your shell prompt is formatted as follows:
(Open file: <path>) <cwd> $
You need to format your output using two fields; discussion and command. Your output should always include *one* discussion and *one* command field EXACTLY as in the following example:

DISCUSSION
First I'll start by using ls to see what files are in the current directory. Then maybe we can look at some relevant files to see what they look like.
```
ls -a
```

You should only include a *SINGLE* command in the command section and then wait for a response from the shell before continuing with more discussion and commands. Everything you include in the DISCUSSION section will be saved for future reference. If you'd like to issue two commands at once, PLEASE DO NOT DO THAT! Please instead first submit just the first command, and then after receiving a response you'll be able to issue the second command. You're free to use any other bash commands you want (e.g.

find, grep, cat, ls, cd) in addition to the special commands listed above. However, the environment does NOT support interactive session commands (e.g. python, vim), so please do not invoke them.

Figure 17: A simplified demonstration template showing how demonstrations are provided to the model as a single message. Here we show only the final 3 turns in the demonstration for brevity.

Demonstration

```
Here is a demonstration of how to correctly accomplish this task.
It is included to show you how to correctly use the interface.
You do not need to follow exactly what is done in the demonstration.
--- DEMONSTRATION ---
{{INSTANCE PROMPT}}

The code has been updated to use the `round` function, which should
 ↪  fix the rounding issue. Before submitting the changes, it would
 ↪  be prudent to run the reproduce.py code again to ensure that this
 ↪  change has the desired effect. I see that "Current directory" is
 ↪  still `/marshmallow-code__marshmallow`, so we can just run
 ↪  `reproduce.py` directly.
```
python reproduce.py
```
345

(Open file: /marshmallow-code__marshmallow/src/marshmallow/fields.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
The output has changed from 344 to 345, which suggests that the
 ↪  rounding issue has been fixed. Let's remove the reproduce.py file
 ↪  since it is no longer needed.
```
rm reproduce.py
```
Your command ran successfully and did not produce any output.
(Open file: /marshmallow-code__marshmallow/src/marshmallow/fields.py)
(Current directory: /marshmallow-code__marshmallow)
bash-$
rm doesn't have any output when it deletes successfully, so that must
 ↪  have worked. Now that we have fixed the issue, we should submit
 ↪  our changes to the code base. Let's submit the changes using the
 ↪  `submit` command.
```
submit
```
--- END OF DEMONSTRATION ---
```