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

John Yang* **Carlos E. Jimenez*** **Alexander Wettig** **Kilian Lieret**
Shunyu Yao **Karthik Narasimhan** **Ofir Press**

Princeton Language and Intelligence, Princeton University

Abstract

Language model (LM) agents are increasingly being used to automate complicated tasks in digital environments. Just as humans benefit from powerful software applications, such as integrated development environments, for complex tasks like software engineering, we posit that LM agents represent a new category of end users with their own needs and abilities, and would benefit from specially-built interfaces to the software they use. We investigate how interface design affects the performance of language model agents. As a result of this exploration, we introduce SWE-agent: a system that facilitates LM agents to autonomously use computers to solve software engineering tasks. SWE-agent's custom agent-computer interface (ACI) significantly enhances an agent's ability to create and edit code files, navigate entire repositories, and execute tests and other programs. We evaluate SWE-agent on SWE-bench and HumanEvalFix, achieving state-of-the-art performance on both with a pass@1 rate of 12.5% and 87.7%, respectively, far exceeding the previous state-of-the-art achieved with non-interactive LMs. Finally, we provide insight on how the design of the ACI can impact agents' behavior and performance.

1 Introduction

Recent work has demonstrated the efficacy of LM agents for code generation with execution feedback [39]. However, applying agents to more complex code tasks like software engineering remains unexplored. To solve programming tasks, LM agents are typically designed to use existing applications, such as the Linux shell or Python interpreter [53, 57, 59]. However, to perform more complex programming tasks such as software engineering [20], human engineers benefit from sophisticated applications like VSCode with powerful tools and extensions. Inspired by human-computer interaction (HCI) studies on the efficacy of user interfaces for humans [7], we investigate whether LM agents could similarly benefit from better-designed interfaces for performing software engineering tasks.

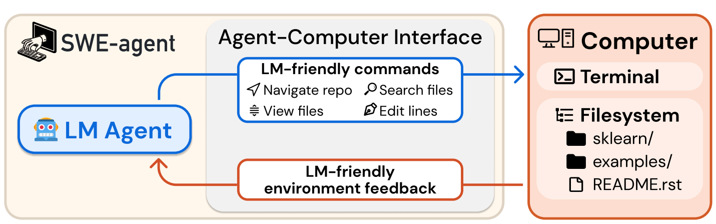

Figure 1: SWE-agent is an LM interacting with a computer through an agent-computer interface (ACI), which includes the commands the agent uses and the format of the feedback from the computer.

*Equal contribution. Correspondence to johnby@stanford.edu, carlosej@princeton.edu.
Data, code, and leaderboard at swe-agent.com

38th Conference on Neural Information Processing Systems (NeurIPS 2024).

Consider the simple setting of an agent interacting directly with a Linux shell [59]. In practice, we find that LM agents can struggle to reliably take actions in this environment. For example, it fails to provide simple commands to edit a small file segment, and does not provide any feedback if the user makes an invalid edit. These deficits substantially hamper performance, motivating the need for an agent-computer interface (ACI), i.e., an abstraction layer between the LM agent and computer, to enhance the LM agent's abilities in computer environments (Figure 1).

From this effort, we introduce SWE-agent, an agent composed of an LM and ACI, that can interact with a computer to solve challenging real-world software engineering problems, such as those proposed in SWE-bench [20]. In contrast to the Linux Shell's granular, highly configurable action space, SWE-agent's ACI instead offers a small set of simple actions for viewing, searching through and editing files. The ACI uses guardrails to prevent common mistakes, and an agent receives specific, concise feedback about a command's effects at every turn. *We show that ACIs tailored specifically for LMs outperform existing user interfaces* (UIs) *designed for human users*, such as the Linux shell.

Using GPT-4 Turbo as a base LM, SWE-agent solves 12.47% of the 2,294 SWE-bench test tasks, substantially outperforming the previous best resolve rate of 3.8% by a non-interactive, retrieval-augmented system [20]. We perform an ablation study on a subset of 300 SWE-bench test instances (SWE-bench Lite) to analyze our ACI design choices. The results show that SWE-agent solves 10.7 percentage points *more* instances than the baseline agent, which uses only the default Linux shell. Although our ACI was developed for GPT-4 Turbo, we show that it is portable to a different LM; SWE-agent with Claude 3 Opus can solve 10.5% of the benchmark tasks.

Our contributions are twofold. First, we introduce the concept of the agent-computer interface (ACI) and demonstrate how careful ACI design can substantially improve LM agent performance without modifying the underlying LM's weights. Second, we build, evaluate, and open-source SWE-agent, a system that provides LMs an ACI for solving real-world software engineering tasks. Unlike prior works that independently explore the merits of tool use, prompting techniques, and code execution in interactive settings, our approach unifies these factors within the ACI framework. We show that crafting LM-centric interactive components has meaningful effects on downstream task performance.

2 The Agent-Computer Interface

An LM acts as an agent when it interacts with an environment by iteratively taking actions and receiving feedback [42, 62]. Typically, the environment has hard constraints, as in robotics, where agents control actuators in the physical world. On the other hand, digital environments can be molded by abstractions in the form of application programming interfaces and user interfaces for software and humans respectively. Naturally, existing interfaces have been designed with one of these users in mind. We argue that LM agents represent a new category of end user, with their own needs and abilities. We refer to the interface LM agents use to interact with computers as the *agent-computer interface* (ACI). Figure 2 illustrates how ACIs provide LM agents with important functionality to interface with computers, similar to how code editors also help humans use computers more effectively.

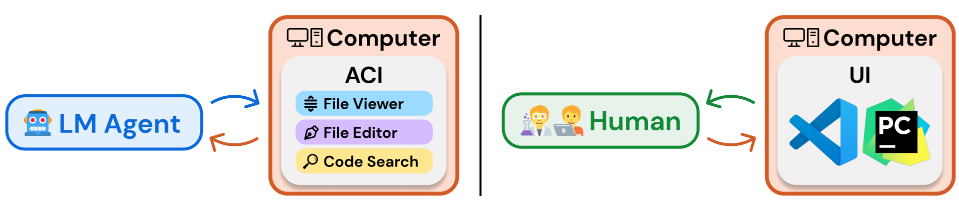

Figure 2: Specialized applications like IDEs (e.g., VSCode, PyCharm) make scientists and software engineers more efficient and effective at computer tasks. Similarly, ACI design aims to create a suitable interface that makes LM agents more effective at digital work such as software engineering.

Disparities in humans' and LMs' abilities and limitations motivates different interface design guidelines. For instance, the current generation of LMs lack the visual understanding abilities to directly operate GUI-based applications with rich visual components and signals. However, many of the features provided by these applications, such as syntax checking and navigation tools, could be useful to LM agents if they were presented in a suitable manner. Additionally, humans can flexibly ignore unnecessary information, whereas all content has a fixed cost in memory and computation for LMs and distracting context can harm performance [27]. Therefore, LM agents may be more effective at interacting with computers when provided an interface that was built informed by these differences.

Ultimately, a well-designed ACI should help the LM agent understand the state of the application given previous changes, manage history to avoid unnecessary context from prior observations, and provide actions that models can use efficiently and reliably. The ACI specifies both the commands available to the LM and how the environment state is 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 LM.

In this paper, we assume a fixed LM and focus on designing the ACI to improve its performance. This means that we shape the actions, their documentation, and environment feedback to complement an LM's limitations and abilities. We draw inspiration from the field of HCI, where user studies elicit insights about how compatible different interfaces are with respect to human intuition and performance [7]. We use two approaches to enhance performance on a development set: (1) manually inspect agent behavior to identify difficulties and propose improvements, and (2) run a grid search to select the best ACI configuration.

Taking these two actions resulted in several insights about design principles that seem especially important for building effective ACIs:

1. **Actions should be simple and easy to understand for agents.** 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 that we describe in Section 3.
2. **Actions should be compact and 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 poor design would therefore have many simple actions that must be composed across multiple turns for a higher order operation to take effect. We show this idea in action in the Editing and Search interface analyses in Section 5.1.
3. **Environment feedback should be informative but concise.** High quality feedback should provide the agent with substantive information about the current environment state (and the effect of the agent's recent actions) without unnecessary details. For instance, when editing a file, updating the agent about revised content is helpful. Figures 3a, 3b and Table 3 show this.
4. **Guardrails mitigate error propagation and hasten recovery.** Like humans, LMs make mistakes when editing or searching and can struggle to recover from these errors. Building in guardrails, such as a code syntax checker that automatically detects mistakes, can help agents recognize and quickly correct errors. We show the effect of editing guardrails in Table 3.

Analysis and ablation studies in Section 5 demonstrate how alternative ACIs affect LM performance. Our studies shows how these principles appear recurrently across actions, feedback, and workflows.

3 SWE-agent: Designing an ACI for Software Engineering

Here we describe how SWE-agent provides an ACI for LMs to act as software engineering agents, enabling them to effectively search, navigate, edit, and execute code commands. The ACI comprises several principal components, including search/navigation, file viewer, file editor, and context management. At each step, SWE-agent generates a thought and a command, then incorporates the feedback from the command's execution in the environment (ReAct; Yao et al. [62]). Built atop the Linux shell, SWE-agent also allows access to common Linux commands and utilities when needed.

Search and navigation. Navigating codebases requires finding the relevant file and content. A common strategy to do this involves looking up terms that might be useful, e.g., files, functions, or class definitions mentioned in an issue. We introduce the special commands `find_file`, `search_file`, and `search_dir`, which output a summary of search results when searching for filenames and strings within files or directories. Figure 10 shows examples of these search result formats. The `find_file` command searches for filenames in the repository, while the `search_file` and `search_dir` locates strings in a file(s) of a subdirectory. Our interface encourages efficient searches by suppressing verbose results. The search commands return at most 50

Observation (File Viewer)

```
[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.

Response (Thought + Edit Action)

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. (a) The file viewer shows the agent the open file's content with line numbers. (b) The agent invokes the edit function to replace lines 404-407 in the open file. After the edit, the file viewer shows the agent the now updated version of the file.

results for each search query; if a search exceeds this number, we do not report the results and instead suggest that the agent write a more specific query.

File viewer. After finding a file they want to view, agents use the interactive file viewer by calling the command `open` on the relevant file path. 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 access 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 visible line). Figure 3a shows an example of this interface.

File editor. We provide a few commands that let LMs 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. This 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, as shown in Figure 3b. After edits are applied, the file viewer automatically displays the updated content, helping the agent observe the effects of its edit immediately without invoking additional commands. Figure 3b shows an example agent response, including a file edit.

Similar to how humans can use tools like syntax highlighting to help them notice format errors when editing files in an IDE, we integrate a code linter into the `edit` function to alert the agent of mistakes it may have introduced when editing a file. Select errors from the linter are shown to the agent along with a snippet of the file contents before/after the error was introduced. Invalid edits are discarded, and the agent 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 demonstrations on the correct use of bash and ACI commands. At each step, the system instructs them to generate both a *thought* and an *action* [62]. Malformed generations trigger an error response, shown in Figure 32, asking the agent to try again, which is repeated until a valid generation is received. Once received, all past error messages except the first are omitted.

The agent's environment responses display computer output using the template shown in Figure 30; however, if no output is generated, a specific message ("Your command ran successfully and did not produce any output") is included to enhance clarity. To further improve context relevance, observations preceding the last 5 are each collapsed into a single line, shown in Figure 31. By removing most content from prior observations, we maintain essential information about the plan and action history while reducing unnecessary context, which allows for more interaction cycles and avoids showing outdated file information. §A provides further implementation details.

4 Experimental Setup

Datasets. We primarily evaluate on the SWE-bench dataset, which includes 2,294 task instances from 12 different repositories of popular Python packages [20]. 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 that focus on evaluating self-contained functional bug fixes. We also test SWE-agent's basic code editing abilities with HumanEvalFix, a short-form code debugging benchmark [32].

Models. All results, ablations, and analyses are based on two leading LMs, GPT-4 Turbo (`gpt-4-1106-preview`) [34] and Claude 3 Opus (`claude-3-opus-20240229`) [6]. We experimented with a number of additional closed and open source models, including Llama 3 and DeepSeek Coder [14], but found their performance in the agent setting to be subpar. Many LMs' context window is too small, such as Llama 3's context window of 8k. GPT-4 Turbo and Claude 3 Opus have 128k and 200k token context windows, respectively, which provides sufficient room for the LM to interact for several turns after being fed the system prompt, issue description, and optionally, a demonstration.

Baselines. We compare SWE-agent to two baselines. The first setting is the non-interactive, retrieval-augmented generation (RAG) baselines established in Jimenez et al. [20]. Here, a BM25 retrieval system retrieves the most relevant codebase files using the issue as the query; given 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. [59]. Following the InterCode environment, this baseline system asks the LM to resolve the issue by interacting with a shell process on Linux. Like SWE-agent, model prediction is generated automatically based on the final state of the codebase after interaction.

Metrics. We report **% Resolved** or **pass**@1 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 [20]. We also report the **$ Avg. Cost** metric, the API inference cost incurred by SWE-agent averaged over all successfully resolved instances. Due to budget constraints, we set the per-instance budget to $4; if a run exceeded this budget, existing edits were submitted automatically.

Configuration search. During the design process of SWE-agent, we arrived at the final ACI design through qualitative analysis of system behavior on a small set of hand-picked examples from the development split of SWE-bench. For the remaining hyperparameter choices, we performed a sweep over the window size, history processing, and decoding temperature, shown in §B.1.

5 Results

Across all systems, SWE-agent w/ GPT-4 Turbo achieves the best performance all-around, successfully solving 12.47% (286/2,294) of the full SWE-bench test set and 18.00% (54/300) of the Lite split. As shown in Table 1, compared to RAG on Lite, SWE-agent is 8-13x more costly but yields a 6.7-fold improved % Resolved rate. An LM-friendly ACI's value is confirmed by SWE-agent's 64% relative increase compared to Shell-only, both with GPT-4 Turbo.

In Table 2, SWE-agent yields strong performance on HumanEvalFix with 88.3% pass@1 rate. Figure 4 reveals that average performance variance is relatively low, but per-instance resolution can change considerably. More results are given in the appendix: §B.2 shows that the success rate is uncorrelated to the issue age (controlling for possible test pollution), B.5 presents more details on performance variance and pass@k, and B.7 discusses extra evaluation details.

5.1 Analysis of ACI Design

We perform several ablations of the SWE-agent interface, specifically with respect to the SWE-agent w/ GPT-4 configuration, summarized in Table 3. Our case studies shed light on interesting agent behavior along with the impact of different ACI designs.

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 a Linux shell environment. Some exploration patterns (e.g., chains of `cd`, `ls`, `cat`) are extremely inefficient. `grep` or `find` look ups can perform 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.

https://github.com/meta-llama/llama3
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 [20].

	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

Table 2: Pass@1 results on HumanEvalFix [32]. Except for SWE-agent, we use scores as reported in Yu et al. [65].

Model	Python	JS	Java
CodeLLaMa-instruct-13B	29.2	19.5	32.3
GPT-4	47.0	48.2	50.0
DeepseekCoder-CodeAlpaca-6.7B	49.4	51.8	45.1
WaveCoder-DS-6.7B	57.9	52.4	57.3
SWE-agent w/ GPT-4 Turbo	**87.7**	**89.7**	**87.9**

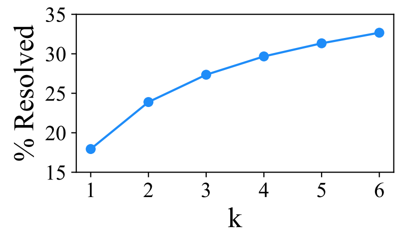

Figure 4: SWE-agent w/ GPT-4 Turbo Pass@k performance across 6 runs on SWE-bench Lite.

Table 3: 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 5 and 6, respectively). We also verify how varying the file viewer window size affects performance, and we ablate the effect of different context management approaches.

Editor		**Search**		**File Viewer**		**Context**	
`edit` action	15.0 $_{\downarrow 3.0}$	Summarized 🤠	18.0	30 lines	14.3 $_{\downarrow 3.7}$	Last 5 Obs. 🤠	18.0
w/ linting 🤠	18.0	Iterative	12.0 $_{\downarrow 6.0}$	100 lines 🤠	18.0	Full history	15.0 $_{\downarrow 3.0}$
No `edit`	10.3 $_{\downarrow 7.7}$	No search	15.7 $_{\downarrow 2.3}$	Full file	12.7 $_{\downarrow 5.3}$	w/o demo.	16.3 $_{\downarrow 1.7}$

Figure 5 compares the Shell-only setting to two different search interfaces. *Iterative* search, directly inspired by traditional user interfaces for search, e.g., `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 even worse performance than the not having additional search tools at all (15.7% $_{\downarrow 2.3}$ for No search vs. 12.0% $_{\downarrow 6.0}$ with Iterative search).

Compact, efficient file editing is critical to performance. SWE-agent's file editor and viewer are designed to consolidate the editing process into a single command that enables easy multi-line edits with consistent feedback and automatically updates the agent's view of the file after editing. In the No `edit` setting, editing options are restrictive and prone to errors; the primary methods available are either replacing entire files through redirection and overwriting or using 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

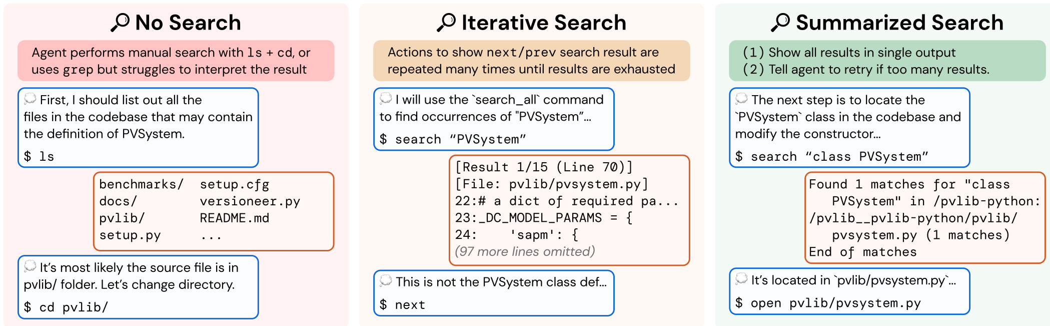

Figure 5: Three different Search interfaces for task instance `pvlib__pvlib-python-1224`. In Shell-only, an agent performs localization using only standard bash commands and utilities. Compared to *Iterative* search, *Summarized* search shows an exhaustive list of search results and provides guidance on refining under-specified queries.

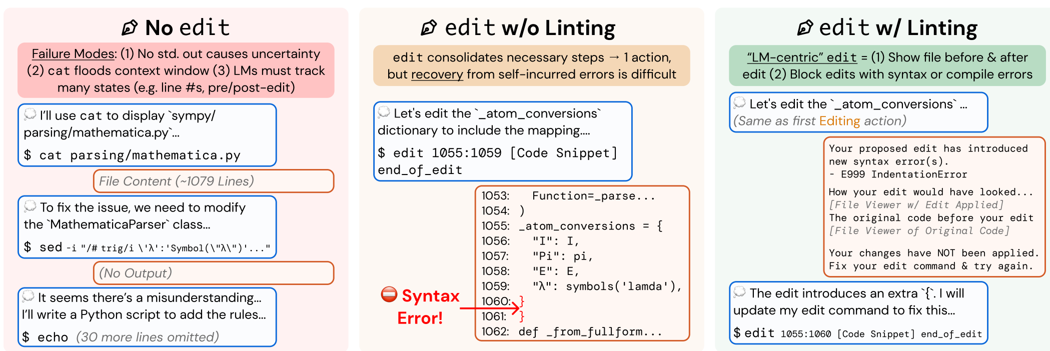

Figure 6: Three different Edit interfaces for task instance `sympy__sympy-24102`. Editing with bash commands requires several actions to successfully modify a file. The *Editing* component defines an `edit` command that leverages the File Viewer component to replace the bash style of editing workflow with a single command. *Linting* is beneficial for stymieing cascading errors that often start with an error-introducing edit by the agent.

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. Either too little content (30 lines, 14.3% $_{\downarrow 3.7}$) or too much (entire file, 12.7% $_{\downarrow 5.3}$) lowers performance.

Guardrails can improve error recovery. A prominent failure mode occurs when models repeatedly `edit` the same code snippet. The usual suspect for this behavior is an agent introducing a syntax error (e.g., incorrect indentation, extra parenthesis) via an errant `edit`. As discussed in Section 3, we add an intervention to the `edit` logic that lets a modification apply only if it does not produce major errors. We compare this interface with the No `edit` and `edit` w/o linting alternatives in Figure 6. This intervention improves performance considerably (without linting, 15.0% $_{\downarrow 3.0}$).

5.2 Analysis of Agent Behavior

Recurring problem-solving patterns emerge when LMs are equipped with a useful, intuitive ACI. We describe 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 begins with either writing reproduction code and/or localizing the issue's cause to specific lines of code. As shown in Figure 7, all trajectories begin with either `create` (reproduction) or `find_file`/`search_dir` (localization). To reproduce, models will `create` a new file, add reproduction code to it with an `edit`, then run with `python`; this is the most popular triple of actions in Table 8. Using this feedback along with file

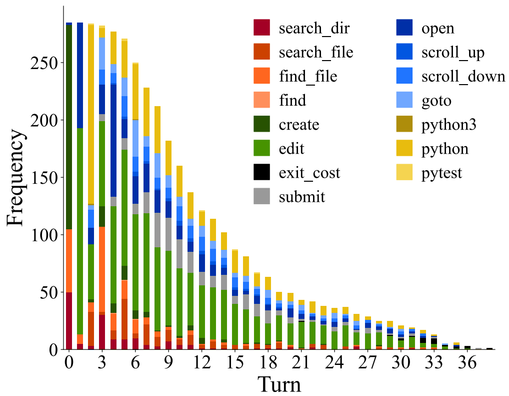

Figure 7: The frequency with which actions are invoked at each turn by SWE-agent w/ GPT-4 for task instances that it solved on the SWE-bench full test set (286 trajectories).

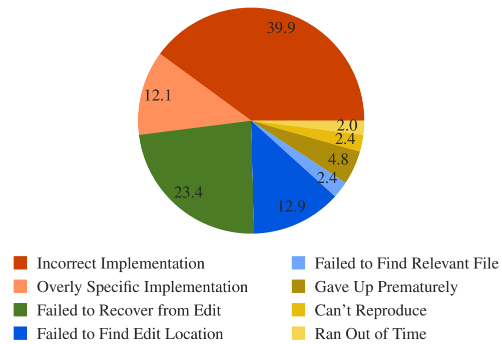

Figure 8: Failure mode distribution for SWE-agent w/ GPT-4 Turbo trajectories of unresolved instances. Each instance is labeled automatically using an LM with the categories from Table 9.

names and symbols in the issue description, an agent will start with a broad, directory-level keyword search, before then zooming into specific files and lines. This is reflected in Figure 22, where the most likely actions following localization sequences like (`python`, `find_file`) and (`search_dir`, `open`) are `search_file` and `goto`, indicative of how an agent "zooms in" on a bug. Extensive analysis on correlations between different groups of actions are discussed in §B.3.3

Remaining turns are mostly "edit, then execute" loops. As exhibited in Figure 7, from turn 5 onwards, the most frequent two actions for all turns are `edit` and `python`. Captured as high probability next actions following (`edit`, `python`) in Figure 22, additional localization operations are often interspersed across these later turns, where agents might look at more in-file code with `search_file`, `scroll_up/down`, or other files altogether with `search_dir`, `find_file`. This behavior usually arises in response to new information from re-running the reproduction script. Submissions are distributed normally from turn 10 onwards, although resolved task instances correlate more with earlier `submits` (see §B.3.1). A walk-through of common trajectory phases is in §B.3.2.

Editing remains challenging for agents. A non-trivial minority of edit actions raise a linting error; out of 2,294 task instances, 1,185 (51.7%) of SWE-agent w/ GPT-4 Turbo trajectories have 1+ failed edits. While agents generally recover more often than not from failed edits, the odds of recovery decrease as the agent accumulates more failed edits. Recovery refers to a sequence of consecutive failed edits followed immediately by a successful edit. Any attempt at editing has a 90.5% chance of eventually being successful. This probability drops off to 57.2% after a single failed edit. More editing phenomena are discussed in §B.3.3, and data about agents' generated fixes are in §B.6.

Agents succeed quickly and fail slowly. We find that runs submitted relatively early are much more likely to be successful compared to those submitted after a larger number of steps or cost. We show in Table 15 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 a cheaper cost than unsuccessful ones. In general, successful instances solved by SWE-agent w/ GPT 4 finish with a median cost of $1.21 and 12 steps compared to a mean of $2.52 and 21 steps for unsuccessful ones. Furthermore, we find that 93.0% of resolved instances are submitted before exhausting their cost budget, compared to 69.0% of instances overall. For these reasons, we suspect that increasing the maximum budget or token limit are unlikely to substantially increase performance. More statistics about how trajectories typically conclude are in §B.9.

Most failures are incorrect implementations. We use GPT-4o to automatically categorize unresolved trajectories (SWE-agent w/ GPT-4 Turbo on SWE-bench Lite, $n = 248$) into one of 9 manually defined categories described in Table 9. On a hand-labeled validation set, the LM's judgment agrees with the authors' on 87% of instances. From Figure 8, about half (52.0%) of unresolved instances fall into the Incorrect Implementation or Overly Specific Implementation categories, suggesting that agents' proposed solutions often simply fail to functionally address the issue or are insufficiently general solutions. Cascading failed edits make up another 23.4% of failures. More details in §B.4.

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 [5, 1, 15, 30]. Subsequent works have built upon the code generation task formulation to contribute new benchmarks that translate problems to different (programming) languages [3, 49], incorporate third-party libraries [25, 29], introduce derivative code completion tasks [18, 32], increase test coverage [26], change the edit scope [8, 9, 64], and add robustness to dataset contamination [19]. 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, the rapid development of LMs has begun to saturate many of these benchmarks. For instance, the top method solves 94.4% of HumanEval [70].

Gauging future trends with the code generation task paradigm can 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 [68, 20, 28]. Repository-level code editing introduces many reasoning challenges grounded in real SE subtasks, such as spotting errant code and 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 tended to frame problems in isolation from the rest of a codebase [21, 23].

We use SWE-bench because it unites many separate SE tasks, such as automated program repair [10, 40, 55], bug localization [4, 58], and testing [22, 46, 56] under a single task formulation that faithfully mirrors practical SE. Furthermore, SWE-bench task instances are diverse, having been automatically collected from real GitHub issues across 12 different repositories. In addition, SWE-bench performance is based on rigorous, execution-based evaluation with human-written unit tests.

6.2 Language Models as Agents

The co-emergence of stronger LMs, increasingly challenging benchmarks, and practical use cases have together motivated a paradigm shift in LMs' inference setting. Instead of traditional zero/few-shot generation, LM agents [17, 42, 47, 54] that interact with a real/virtual world have proliferated as the default setting for web navigation [24, 33, 36, 41, 45, 61, 62, 71], computer control [35, 53, 57], and code generation tasks [16, 50, 63].

Interaction and code generation are increasingly used together, with code as the modality of choice for actions [48, 59], tool construction [13, 51, 69], and reasoning [39, 66, 67]. Coding agents have also been applied to offensive security [11, 37, 60], theorem proving [44], and clinical tasks [38, 43, 52]. To the best of our knowledge, SWE-agent is the first work to explore language agents for end-to-end software engineering (SE).

7 Discussion

We introduce SWE-agent, an agent composed of an LM and ACI capable of autonomously solving software engineering tasks. Through our design methodology, results, and analysis, we demonstrate the value of ACIs tailored to leverage LMs' strengths and mitigate their weaknesses. Beyond empirical applications, we hope the further study of ACIs can also make principled use of and contribute to our understanding of language models and agents, analogous to the synergy between human-computer interaction (HCI) and psychology [2]. Humans and LMs have different characteristics, training objectives, specialities, and limitations [12, 31], and the interaction design processes can be seen as systematic behavioral experimentation that could reveal more insights into these differences towards establishing 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, Alisa Liu, Ori Yoran 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

[1] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, and C. Sutton. Program synthesis with large language models, 2021.

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

[3] F. Cassano, J. Gouwar, D. Nguyen, S. Nguyen, L. Phipps-Costin, D. Pinckney, M.-H. Yee, Y. Zi, C. J. Anderson, M. Q. Feldman, A. Guha, M. Greenberg, and A. Jangda. Multipl-e: A scalable and extensible approach to benchmarking neural code generation, 2022.

[4] S. Chakraborty, Y. Li, M. Irvine, R. Saha, and B. Ray. Entropy guided spectrum based bug localization using statistical language model. *arXiv preprint arXiv:1802.06947*, 2018.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, and J. K. et. al. Evaluating large language models trained on code, 2021.

[6] W.-L. Chiang, L. Zheng, Y. Sheng, A. N. Angelopoulos, T. Li, D. Li, H. Zhang, B. Zhu, M. Jordan, J. E. Gonzalez, and I. Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024.

[7] A. Cooper, R. Reimann, and D. Cronin. *About face 3: the essentials of interaction design*. John Wiley & Sons, Inc., USA, 2007. ISBN 9780470084113.

[8] Y. Ding, Z. Wang, W. U. Ahmad, H. Ding, M. Tan, N. Jain, M. K. Ramanathan, R. Nallapati, P. Bhatia, D. Roth, and B. 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.

[9] X. Du, M. Liu, K. Wang, H. Wang, J. Liu, Y. Chen, J. Feng, C. Sha, X. Peng, and Y. Lou. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation, 2023.

[10] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large language models, 2023.

[11] R. Fang, R. Bindu, A. Gupta, Q. Zhan, and D. Kang. Llm agents can autonomously hack websites, 2024.

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

[13] Y. Gu, Y. Shu, H. Yu, X. Liu, Y. Dong, J. Tang, J. Srinivasa, H. Latapie, and Y. Su. Middleware for llms: Tools are instrumental for language agents in complex environments, 2024.

[14] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *CoRR*, abs/2401.14196, 2024. URL https://arxiv.org/abs/2401.14196.

[15] D. Hendrycks, S. Basart, S. Kadavath, M. Mazeika, A. Arora, E. Guo, C. Burns, S. Puranik, H. He, D. Song, and J. Steinhardt. Measuring coding challenge competence with apps, 2021.

[16] S. Holt, M. R. Luyten, and M. van der Schaar. L2MAC: Large language model automatic computer for unbounded code generation. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=EhrzQwsV4K.

[17] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, C. Zhang, J. Wang, Z. Wang, S. K. S. Yau, Z. Lin, L. Zhou, C. Ran, L. Xiao, C. Wu, and J. Schmidhuber. Metagpt: Meta programming for a multi-agent collaborative framework, 2023.

[18] Q. Huang, J. Vora, P. Liang, and J. Leskovec. Mlagentbench: Evaluating language agents on machine learning experimentation, 2024.

[19] N. Jain, K. Han, A. Gu, W.-D. Li, F. Yan, T. Zhang, S. Wang, A. Solar-Lezama, K. Sen, and I. Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024.

[20] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

[21] R. Just, D. Jalali, and M. 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*, pages 437–440, San Jose, CA, USA, July 2014. Tool demo.

[22] S. Kang, J. Yoon, and S. Yoo. Large language models are few-shot testers: Exploring llm-based general bug reproduction, 2023.

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

[24] J. Y. Koh, R. Lo, L. Jang, V. Duvvur, M. C. Lim, P.-Y. Huang, G. Neubig, S. Zhou, R. Salakhutdinov, and D. Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks, 2024.

[25] Y. Lai, C. Li, Y. Wang, T. Zhang, R. Zhong, L. Zettlemoyer, S. W. tau Yih, D. Fried, S. Wang, and T. Yu. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.

[26] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023.

[27] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang. Lost in the middle: How language models use long contexts, 2023.

[28] T. Liu, C. Xu, and J. McAuley. Repobench: Benchmarking repository-level code auto-completion systems. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=pPjZIOuQuF.

[29] Y. Liu, X. Tang, Z. Cai, J. Lu, Y. Zhang, Y. Shao, Z. Deng, H. Hu, K. An, R. Huang, S. Si, S. Chen, H. Zhao, L. Chen, Y. Wang, T. Liu, Z. Jiang, B. Chang, Y. Qin, W. Zhou, Y. Zhao, A. Cohan, and M. Gerstein. Ml-bench: Evaluating large language models for code generation in repository-level machine learning tasks, 2024.

[30] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, and S. Liu. Codexglue: A machine learning benchmark dataset for code understanding and generation, 2021.

[31] R. T. McCoy, S. Yao, D. Friedman, M. Hardy, and T. L. Griffiths. Embers of autoregression: Understanding large language models through the problem they are trained to solve. *arXiv preprint arXiv:2309.13638*, 2023.

[32] N. Muennighoff, Q. Liu, A. R. Zebaze, Q. Zheng, B. Hui, T. Y. Zhuo, S. Singh, X. Tang, L. V. Werra, and S. Longpre. Octopack: Instruction tuning code large language models. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=mw1PWNSWZP.

[33] R. Nakano, J. Hilton, S. Balaji, J. Wu, L. Ouyang, C. Kim, C. Hesse, S. Jain, V. Kosaraju, W. Saunders, X. Jiang, K. Cobbe, T. Eloundou, G. Krueger, K. Button, M. Knight, B. Chess, and J. Schulman. Webgpt: Browser-assisted question-answering with human feedback, 2022.

[34] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, R. Avila, I. Babuschkin, S. Balaji, V. Balcom, P. Baltescu, H. Bao, M. Bavarian, J. Belgum, I. Bello, J. Berdine, G. Bernadett-Shapiro, C. Berner, L. Bogdonoff, O. Boiko, M. Boyd, A.-L. Brakman, G. Brockman, T. Brooks, M. Brundage, K. Button, T. Cai, R. Campbell, A. Cann, B. Carey, C. Carlson, R. Carmichael, B. Chan, C. Chang, F. Chantzis, D. Chen, S. Chen, R. Chen, J. Chen, M. Chen, B. Chess, C. Cho, C. Chu, H. W. Chung, D. Cummings, J. Currier, Y. Dai, C. Decareaux, T. Degry, N. Deutsch, D. Deville, A. Dhar, D. Dohan, S. Dowling, S. Dunning, A. Ecoffet, A. Eleti, T. Eloundou, D. Farhi, L. Fedus, N. Felix, S. P. Fishman, J. Forte, I. Fulford, L. Gao, E. Georges, C. Gibson, V. Goel, T. Gogineni, G. Goh, R. Gontijo-Lopes, J. Gordon, M. Grafstein, S. Gray, R. Greene, J. Gross, S. S. Gu, Y. Guo, C. Hallacy, J. Han, J. Harris, Y. He, M. Heaton, J. Heidecke, C. Hesse, A. Hickey, W. Hickey, P. Hoeschele, B. Houghton, K. Hsu, S. Hu, X. Hu, J. Huizinga, S. Jain, S. Jain, J. Jang, A. Jiang, R. Jiang, H. Jin, D. Jin, S. Jomoto, B. Jonn, H. Jun, T. Kaftan, Łukasz Kaiser, A. Kamali, I. Kanitscheider, N. S. Keskar, T. Khan, L. Kilpatrick, J. W. Kim, C. Kim, Y. Kim, J. H. Kirchner, J. Kiros, M. Knight, D. Kokotajlo, Łukasz Kondraciuk, A. Kondrich, A. Konstantinidis, K. Kosic, G. Krueger, V. Kuo, M. Lampe, I. Lan, T. Lee, J. Leike, J. Leung, D. Levy, C. M. Li, R. Lim, M. Lin, S. Lin, M. Litwin, T. Lopez, R. Lowe, P. Lue, A. Makanju, K. Malfacini, S. Manning, T. Markov, Y. Markovski, B. Martin, K. Mayer, A. Mayne, B. McGrew, S. M. McKinney, C. McLeavey, P. McMillan, J. McNeil, D. Medina, A. Mehta, J. Menick, L. Metz, A. Mishchenko, P. Mishkin, V. Monaco, E. Morikawa, D. Mossing, T. Mu, M. Murati, O. Murk, D. Mély, A. Nair, R. Nakano, R. Nayak, A. Neelakantan, R. Ngo, H. Noh, L. Ouyang, C. O'Keefe, J. Pachocki, A. Paino, J. Palermo, A. Pantuliano, G. Parascandolo, J. Parish, E. Parparita, A. Passos, M. Pavlov, A. Peng, A. Perelman, F. de Avila Belbute Peres, M. Petrov, H. P. de Oliveira Pinto, Michael, Pokorny, M. Pokrass, V. H. Pong, T. Powell, A. Power, B. Power, E. Proehl, R. Puri, A. Radford, J. Rae, A. Ramesh, C. Raymond, F. Real, K. Rimbach, C. Ross, B. Rotsted, H. Roussez, N. Ryder, M. Saltarelli, T. Sanders, S. Santurkar, G. Sastry, H. Schmidt, D. Schnurr, J. Schulman, D. Selsam, K. Sheppard, T. Sherbakov, J. Shieh, S. Shoker, P. Shyam, S. Sidor, E. Sigler, M. Simens, J. Sitkin, K. Slama, I. Sohl, B. Sokolowsky, Y. Song, N. Staudacher, F. P. Such, N. Summers, I. Sutskever, J. Tang, N. Tezak, M. B. Thompson, P. Tillet, A. Tootoonchian, E. Tseng, P. Tuggle, N. Turley, J. Tworek, J. F. C. Uribe, A. Vallone, A. Vijayvergiya, C. Voss, C. Wainwright, J. J. Wang, A. Wang, B. Wang, J. Ward, J. Wei, C. Weinmann, A. Welihinda, P. Welinder, J. Weng, L. Weng, M. Wiethoff, D. Willner, C. Winter, S. Wolrich, H. Wong, L. Workman, S. Wu, J. Wu, M. Wu, K. Xiao, T. Xu, S. Yoo, K. Yu, Q. Yuan, W. Zaremba, R. Zellers, C. Zhang, M. Zhang, S. Zhao, T. Zheng, J. Zhuang, W. Zhuk, and B. Zoph. Gpt-4 technical report, 2023.

[35] C. Packer, S. Wooders, K. Lin, V. Fang, S. G. Patil, I. Stoica, and J. E. Gonzalez. Memgpt: Towards llms as operating systems, 2024.

[36] O. Press, M. Zhang, S. Min, L. Schmidt, N. Smith, and M. Lewis. Measuring and narrowing the compositionality gap in language models. In H. Bouamor, J. Pino, and K. Bali, editors, *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711, Singapore, Dec. 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.findings-emnlp.378. URL https://aclanthology.org/2023.findings-emnlp.378.

[37] M. Shao, B. Chen, S. Jancheska, B. Dolan-Gavitt, S. Garg, R. Karri, and M. Shafique. An empirical evaluation of llms for solving offensive security challenges, 2024.

[38] W. Shi, R. Xu, Y. Zhuang, Y. Yu, J. Zhang, H. Wu, Y. Zhu, J. Ho, C. Yang, and M. D. Wang. Ehragent: Code empowers large language models for few-shot complex tabular reasoning on electronic health records, 2024.

[39] N. Shinn, F. Cassano, E. Berman, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning, 2023.

[40] D. Sobania, M. Briesch, C. Hanna, and J. Petke. An analysis of the automatic bug fixing performance of chatgpt, 2023.

[41] A. Sridhar, R. Lo, F. F. Xu, H. Zhu, and S. Zhou. Hierarchical prompting assists large language model on web navigation, 2023.

[42] T. Sumers, S. Yao, K. Narasimhan, and T. L. Griffiths. Cognitive architectures for language agents, 2023.

[43] X. Tang, A. Zou, Z. Zhang, Z. Li, Y. Zhao, X. Zhang, A. Cohan, and M. Gerstein. Medagents: Large language models as collaborators for zero-shot medical reasoning, 2024.

[44] A. Thakur, G. Tsoukalas, Y. Wen, J. Xin, and S. Chaudhuri. An in-context learning agent for formal theorem-proving, 2024.

[45] R. Thoppilan, D. D. Freitas, J. Hall, N. Shazeer, A. Kulshreshtha, H.-T. Cheng, A. Jin, T. Bos, L. Baker, Y. Du, Y. Li, H. Lee, H. S. Zheng, A. Ghafouri, M. Menegali, Y. Huang, M. Krikun, D. Lepikhin, J. Qin, D. Chen, Y. Xu, Z. Chen, A. Roberts, M. Bosma, V. Zhao, Y. Zhou, C.-C. Chang, I. Krivokon, W. Rusch, M. Pickett, P. Srinivasan, L. Man, K. Meier-Hellstern, M. R. Morris, T. Doshi, R. D. Santos, T. Duke, J. Soraker, B. Zevenbergen, V. Prabhakaran, M. Diaz, B. Hutchinson, K. Olson, A. Molina, E. Hoffman-John, J. Lee, L. Aroyo, R. Rajakumar, A. Butryna, M. Lamm, V. Kuzmina, J. Fenton, A. Cohen, R. Bernstein, R. Kurzweil, B. Aguera-Arcas, C. Cui, M. Croak, E. Chi, and Q. Le. Lamda: Language models for dialog applications, 2022.

[46] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang. Software testing with large language model: Survey, landscape, and vision, 2023.

[47] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6), Mar. 2024. ISSN 2095-2236. doi: 10.1007/s11704-024-40231-1. URL http://dx.doi.org/10.1007/s11704-024-40231-1.

[48] X. Wang, Y. Chen, L. Yuan, Y. Zhang, Y. Li, H. Peng, and H. Ji. Executable code actions elicit better llm agents, 2024.

[49] Z. Wang, G. Cuenca, S. Zhou, F. F. Xu, and G. Neubig. Mconala: A benchmark for code generation from multiple natural languages, 2023.

[50] Z. Wang, S. Zhou, D. Fried, and G. Neubig. Execution-based evaluation for open-domain code generation, 2023.

[51] Z. Wang, D. Fried, and G. Neubig. Trove: Inducing verifiable and efficient toolboxes for solving programmatic tasks, 2024.

[52] M. Wornow, A. Narayan, K. Opsahl-Ong, Q. McIntyre, N. H. Shah, and C. Re. Automating the enterprise with foundation models, 2024.

[53] Z. Wu, C. Han, Z. Ding, Z. Weng, Z. Liu, S. Yao, T. Yu, and L. Kong. Os-copilot: Towards generalist computer agents with self-improvement, 2024.

[54] Z. Xi, W. Chen, X. Guo, W. He, Y. Ding, B. Hong, M. Zhang, J. Wang, S. Jin, E. Zhou, R. Zheng, X. Fan, X. Wang, L. Xiong, Y. Zhou, W. Wang, C. Jiang, Y. Zou, X. Liu, Z. Yin, S. Dou, R. Weng, W. Cheng, Q. Zhang, W. Qin, Y. Zheng, X. Qiu, X. Huang, and T. Gui. The rise and potential of large language model based agents: A survey, 2023.

[55] C. S. Xia and L. 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*, pages 959–971, 2022.

[56] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang. Universal fuzzing via large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, 2023.

[57] T. Xie, D. Zhang, J. Chen, X. Li, S. Zhao, R. Cao, T. J. Hua, Z. Cheng, D. Shin, F. Lei, Y. Liu, Y. Xu, S. Zhou, S. Savarese, C. Xiong, V. Zhong, and T. Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024.

[58] A. Z. H. Yang, C. Le Goues, R. Martins, and V. 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.

[59] J. Yang, A. Prabhakar, K. R. Narasimhan, and S. Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*, 2023. URL https://openreview.net/forum?id=fvKaLF1ns8.

[60] J. Yang, A. Prabhakar, S. Yao, K. Pei, and K. R. Narasimhan. Language agents as hackers: Evaluating cybersecurity skills with capture the flag. In *Multi-Agent Security Workshop@ NeurIPS'23*, 2023.

[61] S. Yao, H. Chen, J. Yang, and K. Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023.

[62] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. R. Narasimhan, and Y. Cao. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*, 2023. URL https://openreview.net/forum?id=WE_vluYUL-X.

[63] P. Yin, W.-D. Li, K. Xiao, A. Rao, Y. Wen, K. Shi, J. Howland, P. Bailey, M. Catasta, H. Michalewski, A. Polozov, and C. Sutton. Natural language to code generation in interactive data science notebooks, 2022.

[64] H. Yu, B. Shen, D. Ran, J. Zhang, Q. Zhang, Y. Ma, G. Liang, Y. Li, T. Xie, and Q. 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.

[65] Z. Yu, X. Zhang, N. Shang, Y. Huang, C. Xu, Y. Zhao, W. Hu, and Q. Yin. Wavecoder: Widespread and versatile enhanced instruction tuning with refined data generation. *arXiv preprint arXiv:2312.14187*, 2023.

[66] E. Zelikman, Q. Huang, G. Poesia, N. D. Goodman, and N. Haber. Parsel: Algorithmic reasoning with language models by composing decompositions, 2022. URL https://arxiv.org/abs/2212.10561.

[67] E. Zelikman, E. Lorch, L. Mackey, and A. T. Kalai. Self-taught optimizer (stop): Recursively self-improving code generation, 2024.

[68] F. Zhang, B. Chen, Y. Zhang, J. Keung, J. Liu, D. Zan, Y. Mao, J.-G. Lou, and W. 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.

[69] S. Zhang, J. Zhang, J. Liu, L. Song, C. Wang, R. Krishna, and Q. Wu. Training language model agents without modifying language models, 2024.

[70] A. Zhou, K. Yan, M. Shlapentokh-Rothman, H. Wang, and Y.-X. Wang. Language agent tree search unifies reasoning acting and planning in language models, 2023.

[71] S. Zhou, F. F. Xu, H. Zhu, X. Zhou, R. Lo, A. Sridhar, X. Cheng, Y. Bisk, D. Fried, U. Alon, and G. Neubig. Webarena: A realistic web environment for building autonomous agents, 2023.