LIVE-SWE-AGENT: Can Software Engineering Agents Self-Evolve on the Fly?

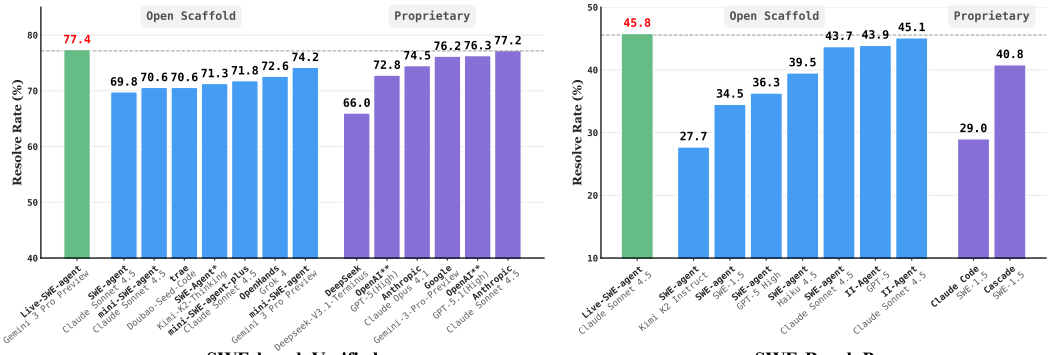
Chunqiu Steven Xia Zhe Wang Yan Yang[†] Yuxiang Wei Lingming Zhang

University of Illinois Urbana-Champaign

{chunqiu2, zhe36, ywei40, lingming}@illinois.edu [†]yanyang826@outlook.com

Abstract

Large Language Models (LLMs) are reshaping almost all industries, including software engineering. In recent years, a number of LLM agents have been proposed to solve real-world software problems. Such software agents are typically equipped with a suite of coding tools and can autonomously decide the next actions to form complete trajectories to solve end-to-end software tasks. While promising, they typically require dedicated design and may still be suboptimal, since it can be extremely challenging and costly to exhaust the entire agent scaffold design space. Recognizing that software agents are inherently software themselves that can be further refined/modified, researchers have proposed a number of self-improving software agents recently, including the Darwin-Gödel Machine (DGM). Meanwhile, such self-improving agents require costly offline training on specific benchmarks and may not generalize well across different LLMs or benchmarks. In this paper, we propose LIVE-SWE-AGENT, the first *live* software agent that can autonomously and continuously evolve itself *on-the-fly* during runtime when solving real-world software problems. More specifically, LIVE-SWE-AGENT starts with the most basic agent scaffold with only access to `bash` tools (e.g., mini-SWE-agent), and autonomously evolves its own scaffold implementation while solving real-world software problems. Our evaluation on the widely studied SWE-bench Verified benchmark shows that LIVE-SWE-AGENT can achieve an impressive solve rate of 77.4% without test-time scaling, outperforming all existing software agents, including the best proprietary solution. Moreover, LIVE-SWE-AGENT outperforms state-of-the-art manually crafted software agents on the recent SWE-Bench Pro benchmark, achieving the best-known solve rate of 45.8%. More details at: https://github.com/OpenAutoCoder/live-swe-agent



*Customized SWE-agent [2] **JSON-based `apply_patch` tool plus dynamic thinking time [1]

Figure 1: SWE-bench Verified and SWE-Bench Pro results (single attempt w/o test-time scaling)

[†]Work done as a research intern at the University of Illinois Urbana-Champaign.

1 Introduction

Large Language Models (LLMs) have rapidly progressed from simple code autocompletion [7, 12, 22, 38] to interactive agents capable of navigating repositories, running tests, and submitting patches end-to-end [42, 34, 47, 37, 10, 23]. Early conversational repair systems leveraged environment feedback to iteratively refine candidate fixes [40], while subsequent agentic frameworks (such as SWE-agent [42] and OpenHands [34]) augmented LLMs with terminals, editors, and search, enabling multi-step tool use on complex repositories. At the same time, complementary "more agentless" pipelines (e.g., Moatless [49] and Agentless [37]) argue that much of the perceived complexity in agent scaffold design can be replaced by dedicated prompting and workflow design.

Despite this progress, most existing agents have fixed designs and are limited in their static action space, where the scaffold implementation (including tools) is preset even when a task could benefit from more customization. In addition, manually designing an optimal software agent scaffold is extremely challenging and costly due to the infinite design space. As a result, more recently, the community has begun to explore self-evolving software agents [45, 30, 33], which iteratively modify their own scaffold implementations and empirically validate each change using offline evaluation signals from coding benchmarks. However, these approaches add significant additional cost. For example, a single run of DGM on SWE-bench costs around $22,000 according to the original paper [45]. Furthermore, they rely heavily on offline evolution, where improvements are typically learned on certain benchmarks and then baked into a static agent. In this way, the learned agents may become specialized to the given benchmarks and underlying LLMs, and may not generalize well beyond that.

To bridge this gap, we introduce LIVE-SWE-AGENT, the first *live*, runtime self-evolving software engineering agent that expands and revises its own capabilities *on the fly* while working on a real-world issue. Our key insight is that software agents are themselves software systems, and modern LLM-based software agents already possess the intrinsic capability to extend or modify their own implementation at runtime. While our idea of enabling on-the-fly self-evolution applies to all parts of the agent scaffold implementation, as a first step, we focus primarily on tool creation, as it is one of the most essential parts of a software agent. LIVE-SWE-AGENT starts from a simplistic agent with only bash tool access (e.g., mini-SWE-agent [42]). During the regular issue-solving loop, the agent can synthesize, modify, and execute custom tools such as editors, code search utilities, and domain-specific analyzers. A lightweight step-reflection prompt repeatedly asks the agent whether creating or revising a tool would accelerate progress, thereby turning tooling into a first-class decision alongside ordinary actions like running tests. This mechanism leaves the underlying scaffold unchanged, requires no offline training, and is agnostic to the underlying LLM. By elevating tool creation to an explicit, iterative decision point, we unlock this hidden *on-the-fly* self-improvement ability.

LIVE-SWE-AGENT addresses the limitations of existing research. First, with tool creation, the agent action space adapts to the problem at hand, producing task-relevant tools that precisely capture the needs to accomplish the current task. Furthermore, by shifting improvement from offline training to online evolution, LIVE-SWE-AGENT alleviates tedious scaffold engineering because new abilities emerge from the encountered issues themselves. Importantly, tool synthesis is not a one-shot preprocessing action but a continuous iterative process interleaved with problem solving. The agent can refine tools as their understanding of the failure mode evolves. Despite its minimal and simplistic design, LIVE-SWE-AGENT is generalizable to different agent scaffolds and LLMs, with state-of-the-art open results on software issue solving.

We have evaluated LIVE-SWE-AGENT on the widely used SWE-bench Verified benchmark [26] and the more challenging SWE-Bench Pro [4]. Without any test-time scaling, LIVE-SWE-AGENT achieves **77.4%** resolve rate on Verified and **45.8%** on Pro, surpassing state-of-the-art open-source baselines and even the best commercial agent systems. Our ablations and tooling analyses reveal that (1) custom tool creation materially improves effectiveness (higher solve rates) with minimal overhead, (2) benefits persist across different state-of-the-art LLM backends, with better results on stronger models, demonstrating its promising future as LLMs' capabilities rapidly evolve, and (3) synthesized tools include both generic utilities and task-aligned specializations that benefit issue-specific problem-solving.

In summary, we make the following contributions:

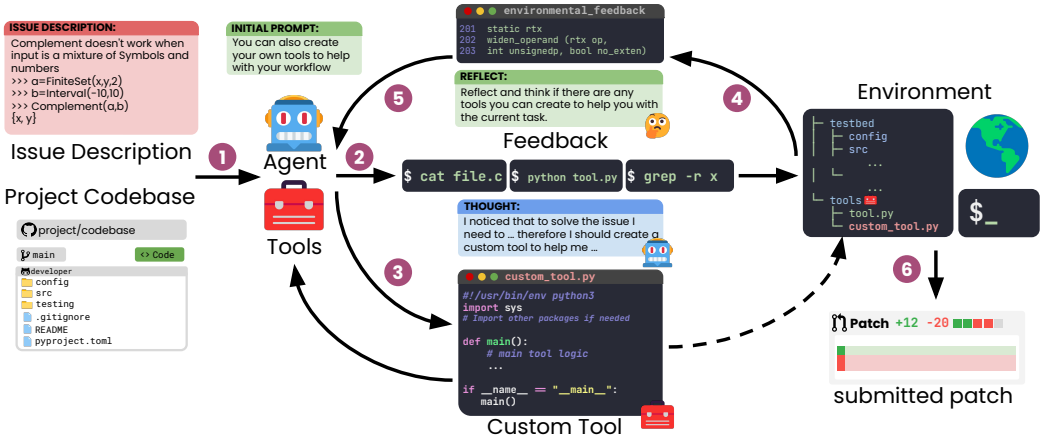


Figure 2: Overview of LIVE-SWE-AGENT

- **The first live software agent.** We present LIVE-SWE-AGENT, the first *live* software agent that can autonomously self-evolve its own scaffold implementation on the fly when solving real-world issues, without any offline training or extra pipelines.
- **Minimal and general implementation.** Our current implementation adopts a minimal and general design, where the agent starts from a minimal bash-only scaffold (e.g., mini-SWE-agent) and self-improves on the fly by creating general or customized tools. The design is compatible with any existing software agent loop or LLM with negligible overhead, has been publicly available at: https://github.com/OpenAutoCoder/live-swe-agent.
- **State-of-the-art performance.** On SWE-bench Verified and SWE-Bench Pro, LIVE-SWE-AGENT reaches 77.4% and 45.8% solve rates (without any test-time scaling), respectively, outperforming all existing open-source agents and commercial systems at the time of writing. To our knowledge, our SWE-Bench Pro result is also the best reported to date.
- **Comprehensive analysis.** We include detailed studies of when and why on-the-fly tool creation helps, how it improves efficiency, and how it compares to other designs like fixed-tool agents, workflow-based systems, and offline self-improving methods. Notably, compared to existing self-improving agents, we achieve much better performance (65.0% solve rate on the SWE-bench Verified-60 subset vs. DGM's 53.3%) while being significantly less costly (no offline cost).
- **Unified leaderboard.** For software tasks, recent LLMs are often benchmarked using manually engineered, proprietary agent scaffolds, making fair model comparison hard. LIVE-SWE-AGENT offers an open, unified, and powerful scaffold that enables genuinely fair, apples-to-apples comparison for future model releases. We are maintaining a leaderboard of recent models evaluated on real-world software tasks using LIVE-SWE-AGENT at: http://live-swe-agent.github.io

2 Approach

LIVE-SWE-AGENT is a live, self-evolving agent that improves and expands its capabilities *on the fly* while solving an issue. Our key insight is recognizing that agents themselves can be iteratively improved, just like the software issues they are designed to solve. The space in which an agent can evolve includes not only the tools it uses but also the underlying agent scaffold itself. In this work, we start from a simple agent scaffold and focus primarily on self-evolution for tool creation and usage, as tools represent one of the most critical components of an agent. We now describe how LIVE-SWE-AGENT provides a framework for LLMs to develop and use their own tools on the fly.

Figure 2 presents an overview of LIVE-SWE-AGENT. First, (1) we take in both the project codebase and the description of the issue to be solved. We then provide this information to the agent and initialize it with a set of tools. In the beginning, the agent may only have access to a limited set of tools (e.g., bash commands) and the goal of LIVE-SWE-AGENT is to allow the agent to generate and use its own tools *on the fly* while solving the issue. As the agent is solving the issue, at each step,it can choose to either (2) output a command (e.g., to use a tool) or (3) create a custom tool that can help

it solve the issue. In LIVE-SWE-AGENT, we define a custom tool as a script that can be executed in the environment. This provides an intuitive and straightforward tool-usage interface for the agent, allowing it to output a command to use any custom tool. Next, different from existing approaches that (4) directly provide the environmental feedback output to the agent, (5) we specifically ask the agent to reflect upon the past steps and decide whether a tool should be created in the feedback message. This loop is repeated until the agent has submitted a solution (6) to the initial problem. Unlike prior agentic setups where the set of tools and possible actions available to the agent is fixed, LIVE-SWE-AGENT allows the agent to perform live self-evolution by creating and using custom tools on the fly.

2.1 On-the-fly Self Evolution

The key idea of LIVE-SWE-AGENT is to enable the agent self-improve by modifying its own scaffold, such as creating custom tools based on the problem and previous trajectories. To support tool creation, we apply several simple modifications to the initial prompt of the agent. Specifically, we provide instructions and examples illustrating how a tool should be created and used. More importantly, we indicate to the agent that: (1) the goal of any tools it creates should be to help it better solve the tasks and (2) the created tools can be for any purpose and do not need to be general.

In addition to introducing the ability for agents to create tools through the initial prompt, we also explicitly ask the agent to reflect on its past trajectory to determine if it should create any tools after each step. This is done by appending a simple reflection message after each environmental feedback. In our experiments, we found that this reflection process is necessary to remind the agents to design tools that are useful and specific for the particular issue.

It is important to point out that the modifications made to a regular agentic framework by LIVE-SWE-AGENT are extremely simple (See Appendix D for both the initial prompt and reflection message). LIVE-SWE-AGENT does not make any changes to the agentic loop, impose a particular workflow, or require any costly offline training. Instead, the focus is on allowing and extending the ability for agents to create their own custom tools at runtime to improve performance and reduce manual tool-creation effort. This allows LIVE-SWE-AGENT to be extremely general and applicable across a wide-range of different tasks, LLMs, and agent scaffolds. Furthermore, we also note that software agents, in essence, are also software. As such, they can be modified and updated on the fly by software agents (themselves) no different than any other software repository. In LIVE-SWE-AGENT, we leverage this insight for on-the-fly self-improvement by enabling the agents to create their own custom tools depending on the problem at hand. We next describe the custom tools agents create in more detail.

2.2 Custom Tool Synthesis

In LIVE-SWE-AGENT, we define a custom tool as a script that can be executed in the environment. This allows the agents to both easily create tools (by creating a script file) and use the created tools (by running the script with arguments). We believe this is a general and intuitive interface that is suitable for a wide range of tasks.

Example editing tool. Figure 3a shows an example of a custom tool created by the agent. This is an editing tool that allows the agent to edit a file by replacing, inserting, or deleting code. We see that it contains the necessary tool logic as well as clear instructions on how the tool can be used. Compared with bash commands which can potentially overwhelm the agent with many different arguments and flags, the tools that agents create themselves have a straightforward purpose and are easy to use. Furthermore, in the example, we see that the editing tool also provides relevant feedback messages such as indicating whether the replacement edit was successful or not. This feedback can be critically important to inform the next actions taken by the agents. On the other hand, a bash editing command like `sed` does not indicate the result of the edit such that a replacement operation where the string to be replaced does not exist will produce no warning message but still return a success code. As such, the agent may be misguided into thinking that the edit was successful when, in fact, no changes have been made to the file.

LIVE-SWE-AGENT also encourages agents to create tools that can improve the efficiency of their workflows. For example, a common step in solving an issue is to identify the locations of relevant code snippets in order to understand the root cause. In this case, a custom search tool can be useful for searching code within certain directories and displaying their surrounding context. While it is possible to achieve the same outcome of a custom search tool using a combination of different bash commands

(e.g., `grep`, `find`, `cat`), the agent often needs to take multiple steps to perform the actual search, leading to increased context length and time required to solve an issue. By creating custom tools that can handle complex and multi-step tasks, LIVE-SWE-AGENT can improve both the effectiveness and efficiency of agents.

Example issue-specific tool. In addition to general tools, another benefit of LIVE-SWE-AGENT is its ability to create issue-specific tools. Figure 3b shows an example of a custom tool that is specifically tailored to a particular issue. In this example, the tool analyzes `MARC` files, a file format for publication or text records, and prints them in a human-readable format. The agent can create this tool and use it to display the content of relevant `MARC` files (including binary files) that serve as test cases, helping it better understand the issue and evaluate potential patches. Such functionality cannot be achieved easily using simple bash commands or even general-purpose tools. By enabling the agents to create arbitrary custom tools on the fly, LIVE-SWE-AGENT can generate specialized tools for individual problems, allowing it to solve issues more effectively.

```
Terminal -- /tmp/edit_tool.py
    """ code omitted """
def edit_file(filename, operations):
    """
    Edit a file with specified operations.
    Operations format:
    - replace:<line_range>:<old_text>:<new_text>
    - insert:<line_num>:<text>
    - delete:<line_num>
    """
    """ code omitted """
    if cmd == 'replace':
        if old_text in lines:
            lines = lines.replace(old_text, new_text)
        else:
            print(f"Warning: '{old_text}'
                  not found in {line_range}")
    """ code omitted """
    print(f"Successfully edited {filename}")

if __name__ == "__main__":
    """ code omitted """
```

(a) Edit tool

```
Terminal -- /tmp/marc_analyzer.py
    """ code omitted """
def analyze_marc_file(filepath):
    """Analyze a MARC file and show language-related
fields."""
    from openlibrary.catalog.marc import MarcBinary
    """ code omitted """
        if filepath.endswith('.xml'):
            root = etree.fromstring(data)
            rec = MarcXml(root)
        else:
            rec = MarcBinary(data)
        """ code omitted """
        edition = read_edition(rec)

        print("Edition data extracted:")
        print(f"  Languages: {edition.get('languages',
'NOT FOUND')}")
        print(f"  Title: {edition.get('title',
                          'NOT FOUND')[:50]}...")
    """ code omitted """

if __name__ == "__main__":
    """ code omitted """
```

(b) `MARC` file analyzer tool

Figure 3: Example custom tools

A fair question to ask is *why not generate these tools directly in the beginning?* The reason is that tool creation, similar to manual problem solving, is also an iterative process. We need to understand the problem at hand and during the solving process identify the issues in order to come up with helpful tools in different scenarios. For example, in the `MARC` file example, it is not immediately obvious that we need to create a custom tool specifically to analyze `MARC` files. By creating all the tools in the beginning and applying this fixed set of tools during the entire process, we will lose the opportunity to design custom tools that are helpful in unique situations. Additionally, having access to all possible tools are not necessarily better as they can overwhelm and mislead the agent. Furthermore, the agent may often iterate on or modify the tools it has created which requires runtime modification abilities not available under a fixed tool setup. LIVE-SWE-AGENT provides the ability for agents to automatically synthesize customs tools on the fly without any additional overhead from heavy scaffold modifications or costly offline training updates.

3 Experimental Setup

Implementation. While LIVE-SWE-AGENT is general across different software agent scaffolds, we implement LIVE-SWE-AGENT on top of the popular mini-SWE-agent [42] framework. As such, we retain the hyperparameters used in mini-SWE-agent by default (i.e., maximum step limit of 250 and maximum cost of $3 per issue). We choose mini-SWE-agent to build on since it is not only simplistic (with $\sim$100 lines of code and only accessing bash commands) but also widely used. Unless otherwise stated, we use Claude 4.5 Sonnet (`claude-sonnet-4-5-20250929`) [6] in our experiments and sample one patch per issue.

Datasets. We evaluate LIVE-SWE-AGENT on the popular SWE-bench Verified benchmark [26] containing 500 software development problems where the goal is to successfully modify the repository given a problem description. SWE-bench Verified is validated by human developers to ensure each problem description has sufficient amount of information to solve the issue. Additionally, we also evaluate on the recent SWE-Bench Pro [4] benchmark, containing 731 publicly available problems, aimed to capture realistic, complex, and enterprise-level problems. Compared with SWE-bench Verified, SWE-Bench Pro contains more difficult problems across multiple repositories and programming languages.

Table 1: Result on SWE-bench Verified

Tool	LLM	% Resolved	Avg. $ Cost
mini-SWE-agent [42]	GPT-5-Mini	59.8%	$0.04
	GPT-5	65.0%	$0.28
	Claude 4.5 Sonnet	70.6%	$0.56
	Gemini 3 Pro	74.2%	$0.46
LIVE-SWE-AGENT	GPT-5-Mini	63.0%	$0.05
	GPT-5	68.4%	$0.27
	Claude 4.5 Sonnet	75.4%	$0.68
	Gemini 3 Pro	77.4%	$0.48

Table 2: Result on SWE-bench Verified-60

	Tool	LLM	% Resolved	Offline cost (hours)
Offline self-improving agents	SICA [30]	GPT-5-Mini	50.0%	infinite loop
	DGM [45]	GPT-5-Mini	53.3%	1231
	HGM [33]	GPT-5-Mini	56.7%	512
	LIVE-SWE-AGENT	GPT-5-Mini	65.0%	0

Baselines. We compare LIVE-SWE-AGENT against representative state-of-the-art agentic approaches. For SWE-bench Verified, we compare against mini-SWE-agent as it is one of the most widely-used open-source agentic solutions on SWE-bench tasks with top leaderboard performance. Additionally, it is also a straightforward comparison as we directly build on top of the mini-SWE-agent framework. Furthermore, we compare with prior self-evolving agents: Self-Improving Coding Agent (SICA) [30], Darwin-Gödel Machine (DGM) [45], and Huxley-Gödel Machine (HGM) [33] on a subset of SWE-bench Verified problems. This subset of 60 SWE-bench Verified problems has been used by prior work [33] to specifically evaluate all three self-improving agent baselines. For SWE-Bench Pro, we compare against SWE-agent [42] which is the top-performing approach on the SWE-Bench Pro leaderboard. For each baseline, we directly reuse their experimental results and report their performance, cost, as well as the backend LLM used whenever possible.

4 Evaluation

4.1 Main Results

SWE-bench Verified. Table 1 shows the results of LIVE-SWE-AGENT and prior agent approaches on SWE-bench Verified. We first observe that compared with mini-SWE-agent, across four different LLM backends, LIVE-SWE-AGENT consistently achieves a higher resolve rate. This demonstrates the improvement in performance enabled by allowing agents to create and use their own custom tools on the fly. Furthermore, LIVE-SWE-AGENT is able to achieve this with only a minimal increase in cost compared with the base mini-SWE-agent. In certain cases (e.g., in GPT-5), we even observed slight cost savings where the agent can improve the solving efficiency by replacing complex, multi-turn commands with customs tools that achieve the same functionality.

We also demonstrate the comparison between LIVE-SWE-AGENT and the best-performing agentic tools, including both state-of-the-art open-source solutions and proprietary commercial scaffolds, on SWE-bench Verified (Figure 1). We only report single-attempt results in Figure 1 without any test-time scaling to ensure a fair comparison. We observe that LIVE-SWE-AGENT using Gemini 3 Pro achieves **a solve rate of 77.4% without test-time scaling, outperforming all existing agents on the SWE-bench Verified leaderboard**[1] including state-of-the-art commercial solutions at the time of writing.

[1] `https://www.swebench.com`

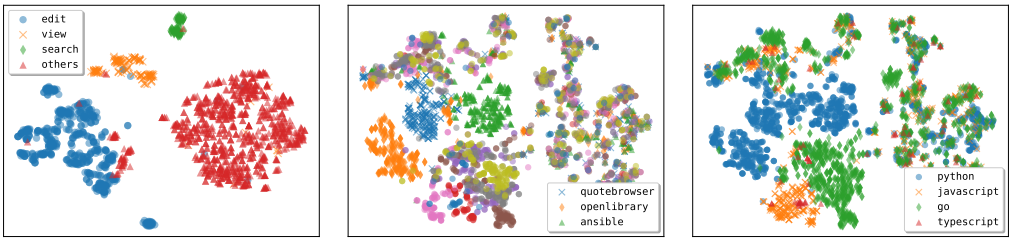
Table 3: Result on SWE-Bench Pro

Tool	LLM	% Resolved	Avg. $ Cost
SWE-agent [42]	Claude 4.5 Sonnet	43.6%	-
LIVE-SWE-AGENT	Claude 4.5 Sonnet	45.8%	$0.73

Furthermore, we perform a detailed comparison with three prior self-evolving agents on a subset of 60 SWE-bench Verified problems chosen by prior evaluation [33]. Table 2 shows the resolve rate and the offline cost for SICA, DGM, HGM, and LIVE-SWE-AGENT on the subset of problems. We observe that LIVE-SWE-AGENT achieves the best performance, improving the resolve rate by 8.3 percentage points compared to previous best approach. We also see that previous self-evolving agents all require a heavy amount of offline training to evolve the base agent, costing more than 500 hours. Furthermore, prior self-evolving techniques produce a static agent used for all problems. On the other hand, LIVE-SWE-AGENT creates customs tools for each individual task, allowing it to adapt on the fly based on the problem and specific LLM used. Unlike the costly offline updates required by prior self-evolving agents, LIVE-SWE-AGENT instead adopts an online evolving approach to prompt the agent to generate custom tools on the fly to improve performance with minimal overhead.

SWE-Bench Pro. We also evaluate LIVE-SWE-AGENT on the public set of SWE-Bench Pro, containing 731 problems across 11 repositories and four programming languages (Python, Go, TypeScript, and JavaScript). Table 3 shows the performance of LIVE-SWE-AGENT compared to the state-of-the-art baseline of SWE-agent on SWE-Bench Pro [4]. Different from mini-SWE-agent, which only has access to basic bash commands, SWE-agent provides handcrafted file viewing and editing tools and is the best performing approach on the SWE-Bench Pro. We choose Claude 4.5 Sonnet for SWE-agent because it ranks at the top of SWE-Bench Pro leaderboard[2], as such, we also use it as the base LLM for LIVE-SWE-AGENT to enable a fair comparison. We observe that LIVE-SWE-AGENT is able to achieve better performance compared with SWE-agent, a specially designed agent implemented with close to 7,000 lines of code. We also compare with all other top-performing agents and LLMs in Figure 1 and observe that LIVE-SWE-AGENT is able to achieve the **new state-of-the-art performance of 45.8% resolve rate on SWE-Bench Pro**. This further demonstrates the superiority of the live scaffold design of LIVE-SWE-AGENT compared to existing agents that interact with a fixed set of manually-crafted tools.

4.2 Tools Analysis



(a) Tools in SWE-bench Verified (b) Tools in SWE-Bench Pro (c) Tools in SWE-Bench Pro

Figure 4: 2 dimensional t-SNE visualization of tools generated by Claude 4.5 Sonnet on SWE-bench Verified and SWE-Bench Pro. We label and display the embedding based on tool type (Figure 4a), repository name (Figure 4b), and programming language used in the repository (Figure 4c). Note that for Figure 4b, we only label three repositories in the legend due to space considerations. The three repositories were chosen as they have representative distinct clusters.

Categories and variations of custom tools. We examine the custom tools created by LIVE-SWE-AGENT. Figure 4 shows the embedding visualizations using t-SNE [17] of the tools created by LIVE-SWE-AGENT. We compute the embedding for each tool based on the tool body (i.e., the

[2] https://scale.com/leaderboard/swe_bench_pro_public

content of the tool script) using a text embedding model (OpenAI `text-embedding-3-small`). To start off, we look at the types of tools created by LIVE-SWE-AGENT by categorizing them into common tool functionalities (e.g., edit, view, search, etc). We perform this categorization by using simple string matching based on the script filename of the tool. Figure 4a shows the visualization of the tools generated for SWE-bench Verified. We see that while there are definitely distinct clusters of common tools like edit, view, and search, there are still variations among them. For example, the edit tool cluster (highlighted in blue circles) is not a tightly concentrated dot, but instead is spreadout, showing that LIVE-SWE-AGENT can generate different editing tools depending on the problem. Furthermore, LIVE-SWE-AGENT also allows the agent to generate additional tools (the others category highlighted in red triangles). These are the more unique issue-specific tools such as a script that applies a very detailed patch to multiple files or a diff checking tool that shows the difference between two files. We also apply similar analysis to all studied datasets in Appendix B.

In addition to the types of tools created, we also examine how the tools created can change across different repositories and languages. Figure 4b shows the visualization of the tools created in SWE-Bench Pro categorized based on repositories. In SWE-Bench Pro, there are 11 different repositories, and not surprisingly, there are common tools created across different repositories (i.e., the overlapping clusters in the figure). On the other hand, there are also specific tools for certain repositories, for example we see a distinct cluster (highlighted in orange diamonds) for `openlibrary`, a codebase for cataloging literature. Since `openlibrary` deals with lots of raw data of a specialized format, the custom tools generated by LIVE-SWE-AGENT are specifically tailored for that. We also observe similar results when we break down the tools based on programming language of the repositories in Figure 4c. Different problems, environments, and languages may require completely different specialized tools, highlighting again how using the same set of tools for all problems is suboptimal.

Effective and interesting custom tools. We now take a closer look at a few exemplary effective and interesting tools created by LIVE-SWE-AGENT. Figure 5a shows a custom search tool created by LIVE-SWE-AGENT. This example might appear simple and straightforward at first glance. However, it implements several important features: (1) supports focused searching within directories and pattern-based matching, (2) ignores irrelevant folders (`__pycache__` and `node_modules`) and hidden folders (`.git`) during search, (3) displays relevant code context before and after the found code, and (4) limits results to top 20 matches to reduce context size. By baking in the exclusion of irrelevant folders in the search tool, agents do not have to add these folders as a separate flag during the searching process. Showing the relevant surrounding context is also helpful since oftentimes we want to find out how the code snippet of interest is being used. Furthermore, only showing the first 20 matches can be critically important to not drastically inflate the context window, allowing the agent to easily refine their search to limit results. An example bash command to emulate the functionality of the custom tool is shown at the top of Figure 5a. We can see the high degree of complexity in terms of options, arguments, and flags used in this long command. On the other hand, we can simply call `"python search_code.py 'code' src/"` to perform the search via the custom tool. By only using basic bash commands, the agent will have to chain together multiple commands using complex flags with potential to incur errors (e.g., older `grep` versions do not support features like `-exclude-dir`) or substantial slowdowns. In fact, in our experimental results, we found that without access to custom tools, the agent may often take multiple steps to accomplish a basic task such as searching for relevant code. This issue is further amplified when steps like search may need to be done multiple times

```
$ grep -RIn --include='*.py' --exclude-dir='.*'\
            --exclude-dir='\_\_pycache\_\_' \
            --exclude-dir='node\_modules' \
            -C2 `code' src/ | head -n 20
```

Equivalent bash command to perform similar search

Terminal -- /tmp/search_tool.py

```
    """ code omitted """
def search_code(pattern, dirs, f_pattern='*.py', context=2):
    # Skip hidden directories and common non-source
directories
    dirs[:] = [d for d in dirs if not d.startswith('.')
               and d not in ['__pycache__', 'node_modules']]
    """ code omitted """
    for filename in files:
        if re.search(pattern, line, re.IGNORECASE):
    """ code omitted """

if __name__ == "__main__":
    print("Usage: search_code.py <pattern> [directory]
                              [file_pattern] [context]")

    matches = search_code(pattern, dir, f_pattern, context)
    """ code omitted """
    for ctx_line in match['context_before']:
        print(f"  {ctx_line.rstrip()}")
    print(f">>> {match['line']}")
    for ctx_line in match['context_after']:
        print(f"  {ctx_line.rstrip()}")
    """ code omitted """
    if len(matches) > 20:
        print(f"\n... and {len(matches) - 20} more matches")
```

(a) Search tool

Terminal -- /tmp/go_analyzer.py

```
    """ code omitted """
def find_struct_definition(content, struct_name):
    """Find and return struct definition with line
numbers."""

    for i, line in enumerate(lines):
        if re.search(rf'type\s+{struct_name}\s+struct\s*{{',
                     line):
    """ code omitted """

def find_function_definition(content, func_name):
    """Find and return function definition with line
numbers."""

def find_all_references(content, identifier):
    """Find all references to an identifier."""

def find_all_imports(content, identifier):
    """Find all imports in file."""

def main():
    print("Usage: go_analyzer.py <command> <file> [args...]")
    """ code omitted """
```

(b) Go file analyzer tool

Figure 5: Example custom tools

Table 4: Ablation result across LIVE-SWE-AGENT setups on subset of SWE-bench Verified problems

Approach	% Resolved	# of tools created
LIVE-SWE-AGENT w/o tool creation	62.0%	0.00
LIVE-SWE-AGENT w/o reflection	64.0%	2.92
LIVE-SWE-AGENT	76.0%	3.28

Table 5: Ablation result across different LLM backends on subset of SWE-bench Verified problems

LLM	mini-SWE-agent	LIVE-SWE-AGENT
GPT-5-Nano	44.0%	14.0% (↓-68.2%)
GPT-5-Mini	60.0%	58.0% (↓-3.3%)
GPT-5	60.0%	68.0% (↑13.3%)
Claude 3.7 Sonnet	46.0%	50.0% (↑8.7%)
Claude 4 Sonnet	58.0%	64.0% (↑10.3%)
Claude 4.5 Sonnet	62.0%	76.0% (↑22.6%)

for one task thus drastically limiting the problem solving efficiency and effectiveness. By creating and introducing efficient and easy to use custom tools designed by agents themselves, LIVE-SWE-AGENT can improve agent performance on complex software engineering tasks.

Figure 5b shows another custom tool `go_analyzer.py` created by LIVE-SWE-AGENT. The tool can be used to analyze a Go file to find any struct and function definition, identifier references, and to obtain the imports used in the file. This example showcases the powerful generative capability of agents where the custom tool can be used as a simple static analyzer for the Go programming language to search and provide key information about a file. Different from the previous search tool example where one can technically mimic the behavior using bash commmands, abeit using very complex chain commands, the tool logic here is even more complex. The custom tool performs strategic pattern matching based on Go grammar and provides an interface to support multiple different usages. Furthermore, by saving this custom tool in an executable script, the agent can reuse the functionality multiple times. By creating and using this custom tool to better understand file structure and content, LIVE-SWE-AGENT is able to solve this issue[3] that prior best performing baseline could not.

4.3 Ablation

To evaluate the effect of different setups of LIVE-SWE-AGENT, we use a random subset of 50 problems in SWE-bench Verified (see Appendix C.1 for the detailed list of problems). Unless otherwise stated, we use Claude 4.5 Sonnet in our ablation experiments.

Effectiveness of tool creation steps. Table 4 shows the impact of different components of LIVE-SWE-AGENT have on performance. We see that by removing the on-the-fly tool creation ability of LIVE-SWE-AGENT (i.e., using the base mini-SWE-agent), we achieve the lowest solve rate. The performance is improved when we indicate to the agent to create custom tools in the initial prompt (row LIVE-SWE-AGENT w/o reflection). However, the highest resolve rate is achieved once we explicitly ask the agent to reflect on the past trajectories to determine if they should create any tools after each step. In our experiments, we found that this reflection process provides a good reminder for the agent to create tools that are specifically designed for the particular issue. Furthermore, while the number of tools do not necessary correlate with solving performance, we also see that this reflection process creates more tools on average compared to only indicating the tool creation in the initial prompt. LIVE-SWE-AGENT is able to improve performance by reflecting on both the current problem and past trajectories to create custom tools on the fly.

Different LLM backends. We now examine the effect of using different LLM backends for LIVE-SWE-AGENT. Table 5 compares the performance of the base mini-SWE-agent and LIVE-SWE-AGENT as we vary the LLM used on the same 50 problems used in the previous ablation experiment. We first observe that there are weaker LLMs for which using LIVE-SWE-AGENT to create custom

[3] `navidrome__navidrome-10108c63c9b5bdf2966ffb3239bbfd89683e37b7`, SWE-Bench Pro

Table 6: Result on a subset of SWE-bench Multilingual problems

Tool	LLM	% Resolved	Avg. $ Cost
mini-SWE-agent [42]	Claude 4.5 Sonnet	40.0%	$0.59
LIVE-SWE-AGENT	Claude 4.5 Sonnet	46.0%	$0.66

tools on the fly even decreases performance. In particular, we see that for GPT-5-Nano, the results are significantly worse when using LIVE-SWE-AGENT compared to base mini-SWE-agent. After examining the trajectories, we found that GPT-5-Nano fails to understand the goal of creating custom tools and is often stuck in a loop thus leading to reduction in performance. This indicates that weaker LLMs may lack the high-level reasoning capabilities to synthesize useful tools on the fly. However, we see that LIVE-SWE-AGENT is able to improve more over the base mini-SWE-agent as we start to use more powerful models. Specifically, state-of-the-art LLMs such as Claude 4 Sonnet and GPT-5 achieves the highest relative resolve rate improvement when using LIVE-SWE-AGENT compared with mini-SWE-agent. This demonstrates the generalizability of LIVE-SWE-AGENT across high-performing LLMs and the potential for LIVE-SWE-AGENT to further improve performance especially as we continue to build more and more powerful LLMs.

SWE-bench Multilingual. In addition to evaluating on SWE-bench Verified and SWE-Bench Pro, we also test the generalizability of LIVE-SWE-AGENT on other benchmarks. We conducted an initial experiment on a subset of 50 problems (see Appendix C.2 for the detailed list of problems) in SWE-bench Multilingual [19], a benchmark of software engineering tasks across 9 programming languages (JavaScript, TypeScript, Rust, Ruby, Go, C/C++, PHP, and Java). Table 6 shows the performance of LIVE-SWE-AGENT compared to mini-SWE-agent on SWE-bench Multilingual. We observed that LIVE-SWE-AGENT achieves a better performance by obtaining a resolve rate of 46.0%, while mini-SWE-agent only has a resolve rate of 40.0%. This demonstrates the generalizability of LIVE-SWE-AGENT on additional challenging problems.

4.4 Discussion and Future Work

Towards general, on the fly self-evolution. In this work, LIVE-SWE-AGENT primarily focuses on enabling agents to self-evolve through custom tool creation and usage. As discussed previously, the key idea of LIVE-SWE-AGENT, to improve and modify the agent itself on the fly, extends beyond creating new tools to also modifying the entire agent implementation. This includes the overall system prompt of the agent, how the agent interacts with the environment, and even the concrete workflow of the agent when it attempts to solve the issue. We believe that the entire agentic loop, much like any other software, can be modified and improved especially on the fly based on feedback and insights gained from solving a problem. Furthermore, we hope to extend the self-evolution loop across different tasks as well. Instead of discarding each evolved agent after a task is completed, we can save and serialize the useful tools and insights (via concepts like Skills [5]) for future tasks. The agent can then load these useful tools and insights obtained while solving previous tasks on the fly to further improve performance and support continuous self-evolution across tasks.

Impact on LLM evaluation. Our experimental results on multiple benchmarks using multiple LLMs demonstrate that LIVE-SWE-AGENT can achieve state-of-the-art performance. This shows that LIVE-SWE-AGENT can provide an effective scaffold for unified evaluation of LLM performance on solving software development issues. Instead of using complex or proprietary agent scaffolds, LIVE-SWE-AGENT provides a simplistic and lightweight approach that can be easily added on top of any agent design to evaluate LLMs. Moreover, LIVE-SWE-AGENT not only evaluates the issue solving ability, but it can also test the tool-creation capability of LLMs and agents. Tools are one of the most important aspect of an agent and directly impacts the issue solving performance. The ability for LLMs to create custom tools, especially for solving complex software development issues, have not been extensively evaluated. To this end, we have already seen some interesting results in our evaluation where weaker LLMs lack high-level reasoning to create useful tools on the fly (see Section 4.3). LIVE-SWE-AGENT provides a unique framework to jointly evaluate both the tool creation and issue resolution ability of LLMs.

Applications beyond software issue resolution tasks. In addition to software issue resolution tasks, LIVE-SWE-AGENT can be easily applied to other challenging software engineering tasks like generating tests [16], detecting and patching vulnerabilities [21], and synthesizing production-ready software from scratch [48]. Compared with issue resolution tasks, other software problem domains will require even more task-specific and diverse tools and agent scaffolds. For example, to detect malicious vulnerabilities in commercial off-the-shelf binaries, we need binary analysis tools and decompilers. To optimize a large complex system, we need to apply profilers and tracing tools. Instead of using a fixed agent scaffold with basic tools that cannot adopt to different problem domains or painstakingly designing specialized agents for each individual task, LIVE-SWE-AGENT can easily generalize to solving tasks in different domains by automatically modifying itself on the fly based on the task at hand.

Self-evolution during LLM training. In this work, LIVE-SWE-AGENT is implemented as a lightweight modification to an existing agent, requiring no offline training or updates. However, the idea of on-the-fly self-evolution can also be easily extended to LLM training, where instead of learning from a fixed workflow and a static set of tools, the LLM learns to also create new tools and modify the scaffold itself during training. In this approach, the resulting LLM will have improved reasoning capabilities as the self-evolving training approach provides additional learning signals, allowing the LLM to solve more complex tasks. Through self-evolving training, LLMs can learn to better create useful, task-specific tools and modify advanced scaffolds dynamically based on the problem at hand. Moreover, the final trained LLM will be compatible with more runtime agent frameworks since it is able to learn from its own created tools and modified scaffolds rather than relying on predefined scaffolds. This adaptability allows it to be more robust and generalize effectively to new scaffold setups and even different tasks.

5 Related Work

5.1 Software Engineering Agents

Inspired by the human debugging process, where developers interact with the environmental feedback (like test failures) and learn from their earlier attempts, ChatRepair [39, 40] proposed the first interactive bug-fixing solution based on LLMs. Since ChatRepair, a large body of research work on bug fixing and general coding tasks aims to automatically provide LLMs more context information through multi-turn conversations [13, 20, 43]. More recently, foundation LLMs have seen substantial advances in their tool use and reasoning capabilities, and it becomes very natural to further equip feedback-driven solutions with such emergent LLM capabilities. In March 2024, Devin AI released the first AI software engineer, which can fully autonomously complete end-to-end software tasks, such as GitHub issue resolution [15]. The initial Devin release showed an impressive resolve rate of 13.86% on SWE-bench [18], a dataset with thousands of real-world GitHub issues. Since then, a large number of dedicated software agent scaffolds have been proposed, including SWE-agent [41], OpenHands [34], AutoCodeRover [47], and Trae Agent [24]. Such software agents typically equip the LLMs with a suite of coding tools and encourage the LLMs to autonomously decide the next actions for completing real-world software tasks. Different from the mainstream software agents, researchers have also proposed various AI software engineer solutions based on pre-defined workflows to challenge the necessity of complicated agent design, such as Agentless [37] and Moatless [49]. Moreover, recent LLMs have increasingly been post-trained on massive real-world software data to better solve software engineering issues, including SWE-RL [36], DeepSWE [25], DeepSeek V3.1 [3], MiniMax M1/M2 [11], Kimi K2 [31], SWE-1.5 [14], and Code World Model (CWM) [10].

Due to the huge design space for software agents, it can be extremely challenging and costly to build an optimal agent scaffold. As a result, a number of self-improving software agents have been proposed very recently, including the Self-Improving Coding Agent (SICA) [30], Darwin-Gödel Machine (DGM) [45], and Huxley-Gödel Machine (HGM) [33]. However, such self-improving agents require costly offline training on known benchmarks, and may not be generalize well across different LLMs, benchmarks, and issue types. Beyond the software engineering domain, prior work [9, 32, 28, 27, 35] explored using LLMs to create tools for general reasoning or embodied tasks, but they do not target real-world software engineering problems. In contrast, in this paper, we propose LIVE-SWE-AGENT, the first live software agent capable of performing practical self-evolution *on-the-fly* during runtime when solving real-world issues. In this way, LIVE-SWE-AGENT requires no offline training at all

and can be easily generalized to different LLMs and issue domains. Moreover, it also demonstrated superior performance compared to all existing self-improving software agents.

5.2 Benchmarks for Software Engineering Agents

To evaluate and demonstrate the performance of software engineering agents, a large number of datasets have been proposed. SWE-bench [18] is one of the earliest and most widely used benchmark datasets for software agents. Besides the initial SWE-bench dataset, which includes thousands of real-world GitHub issues, researchers have also built curated subsets with high-quality and representative issues to support faster and more reliable evaluation, including SWE-bench Lite [18] and SWE-bench Verified [26]. Since SWE-bench mostly focused on Python projects, researchers have proposed a number of SWE-bench-style benchmarks for projects in multiple languages for more comprehensive agent evaluation, including SWE-PolyBench [29], SWE-bench Multilingual [19], and Multi-SWE-bench [44]. Also, SWE-bench relies heavily on manual effort for collecting benchmark instances and setting up executable environments, and is hardly scalable; as a result, researchers have also leveraged software agents to streamline issue collection and environment setup to build live and scalable benchmarks, including SWE-bench-Live [46] and SWE-rebench [8]. More recently, Scale AI has also constructed SWE-Bench Pro [4], which aims to capture more realistic, complex, enterprise-level issues than SWE-bench. For rigorous evaluation of LIVE-SWE-AGENT, our study involved multiple widely used benchmarks, including SWE-bench Verified, SWE-bench Multilingual, SWE-bench Pro.

6 Conclusion

In this paper, we have proposed LIVE-SWE-AGENT, the first *live software agent* that can autonomously and continuously evolve itself *on-the-fly* during runtime when solving real-world software problems. More specifically, LIVE-SWE-AGENT starts with the most basic agent scaffold with only access to `bash` tools, and autonomously evolves its own scaffold implementation while solving real-world software problems. Our evaluation on the widely studied benchmarks (such as SWE-bench Verified and SWE-bench Pro) demonstrated that LIVE-SWE-AGENT outperforms state-of-the-art manually design software agents, demonstrating a promising future for live and self-evolving software agents.

References

[1] GPT-5.1. `https://openai.com/index/gpt-5-1-for-developers/`.

[2] kimi-k2-thinking. `https://moonshotai.github.io/Kimi-K2/thinking.html`.

[3] DeepSeek AI. DeepSeek V3.1. `https://api-docs.deepseek.com/news/news250821`.

[4] Scale AI. Swe-bench pro: Can ai agents solve long-horizon software engineering tasks? *arXiv preprint arXiv:2509.16941*, 2025.

[5] Anthropic. Introducing Agent Skills. `https://www.claude.com/blog/skills`.

[6] Anthropic. Claude Sonnet 4.5, 2025. `https://www.anthropic.com/news/claude-sonnet-4-5`.

[7] 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. *arXiv preprint arXiv:2108.07732*, 2021.

[8] Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. Swe-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. *arXiv preprint arXiv:2505.20411*, 2025.

[9] Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers. In B. Kim, Y. Yue, S. Chaudhuri, K. Fragkiadaki, M. Khan, and Y. Sun, editors, *International Conference on Representation Learning*, volume 2024, pages 54067–54089, 2024.

[10] Quentin Carbonneaux, Gal Cohen, Jonas Gehring, Jacob Kahn, Jannik Kossen, Felix Kreuk, Emily McMilin, Michel Meyer, Yuxiang Wei, David Zhang, et al. Cwm: An open-weights llm for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

[11] Aili Chen, Aonian Li, Bangwei Gong, Binyang Jiang, Bo Fei, Bo Yang, Boji Shan, Changqing Yu, Chao Wang, Cheng Zhu, et al. Minimax-m1: Scaling test-time compute efficiently with lightning attention. *arXiv preprint arXiv:2506.13585*, 2025.

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

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

[14] Cognition. SWE-1.5. `https://cognition.ai/blog/swe-1-5`.

[15] Cognition. Devin AI, 2024. `https://cognition.ai/blog/introducing-devin`.

[16] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*, pages 423–435, 2023.

[17] Geoffrey E Hinton and Sam Roweis. Stochastic neighbor embedding. *Advances in neural information processing systems*, 15, 2002.

[18] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

[19] Kabir Khandpur, Kilian Lieret, Carlos E. Jimenez, Ofir Press, and John Yang. Swe-bench multilingual, 2025. `https://www.swebench.com/multilingual.html`.

[20] Jiaolong Kong, Xiaofei Xie, Mingfei Cheng, Shangqing Liu, Xiaoning Du, and Qi Guo. Contrastrepair: Enhancing conversation-based automated program repair via contrastive test case pairs. *ACM Transactions on Software Engineering and Methodology*, 34(8):1–31, 2025.

[21] Hwiwon Lee, Ziqi Zhang, Hanxiao Lu, and Lingming Zhang. SEC-bench: Automated Benchmarking of LLM Agents on Real-World Software Security Tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems*, 2025.

[22] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *arXiv preprint arXiv:2203.07814*, 2022.

[23] Junwei Liu, Kaixin Wang, Yixuan Chen, Xin Peng, Zhenpeng Chen, Lingming Zhang, and Yiling Lou. Large language model-based agents for software engineering: A survey. *arXiv preprint arXiv:2409.02977*, 2024.

[24] Yizhou Liu, Pengfei Gao, Xinchen Wang, Jie Liu, Yexuan Shi, Zhao Zhang, and Chao Peng. Marscode agent: Ai-native automated bug fixing. *arXiv preprint arXiv:2409.00899*, 2024.

[25] Michael Luo, Naman Jain, Jaskirat Singh, Sijun Tan, Ameen Patel, Qingyang Wu, Alpay Ariyak, Colin Cai, Shang Zhu Tarun Venkat, Ben Athiwaratkun, et al. Deepswe: Training a fully open-sourced, state-of-the-art coding agent by scaling rl.

[26] OpenAI. SWE-bench Verified, 2025. `https://openai.com/index/introducing-swe-bench-verified/`.

[27] Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. Creator: Tool creation for disentangling abstract and concrete reasoning of large language models, 2024.

[28] Jiahao Qiu, Xuan Qi, Tongcheng Zhang, Xinzhe Juan, Jiacheng Guo, Yifu Lu, Yimin Wang, Zixin Yao, Qihan Ren, Xun Jiang, Xing Zhou, Dongrui Liu, Ling Yang, Yue Wu, Kaixuan Huang, Shilong Liu, Hongru Wang, and Mengdi Wang. Alita: Generalist agent enabling scalable agentic reasoning with minimal predefinition and maximal self-evolution, 2025.

[29] Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. Swe-polybench: A multi-language benchmark for repository level evaluation of coding agents. *arXiv preprint arXiv:2504.08703*, 2025.

[30] Maxime Robeyns, Martin Szummer, and Laurence Aitchison. SICA a self-improving coding agent. In *ICLR 2025 Workshop on Scaling Self-Improving Foundation Models*, 2025.

[31] Kimi Team. Kimi k1.5: Scaling reinforcement learning with llms. *arXiv preprint arXiv:2501.12599*, 2025.

[32] Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

[33] Wenyi Wang, Piotr Piękos, Li Nanbo, Firas Laakom, Yimeng Chen, Mateusz Ostaszewski, Mingchen Zhuge, and Jürgen Schmidhuber. Huxley-gödel machine: Human-level coding agent development by an approximation of the optimal self-improving machine, 2025.

[34] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. Openhands: An open platform for ai software developers as generalist agents. *arXiv preprint arXiv:2407.16741*, 2024.

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

[36] Yuxiang Wei, Olivier Duchenne, Jade Copet, Quentin Carbonneaux, Lingming Zhang, Daniel Fried, Gabriel Synnaeve, Rishabh Singh, and Sida I. Wang. Swe-rl: Advancing llm reasoning via reinforcement learning on open software evolution. *arXiv preprint arXiv:2502.18449*, 2025.

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

[38] 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*, pages 959–971, 2022.

[39] Chunqiu Steven Xia and Lingming Zhang. Conversational automated program repair. *arXiv preprint arXiv:2301.13246*, 2023.

[40] Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831, 2024.

[41] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[42] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik R Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

[43] Zhiqiang Yuan, Mingwei Liu, Shiji Ding, Kaixin Wang, Yixuan Chen, Xin Peng, and Yiling Lou. Evaluating and improving chatgpt for unit test generation. *Proceedings of the ACM on Software Engineering*, 1(FSE):1703–1726, 2024.

[44] Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-swe-bench: A multilingual benchmark for issue resolving. *arXiv preprint arXiv:2504.02605*, 2025.

[45] Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin godel machine: Open-ended evolution of self-improving agents. *arXiv preprint arXiv:2505.22954*, 2025.

[46] Linghao Zhang, Shilin He, Chaoyun Zhang, Yu Kang, Bowen Li, Chengxing Xie, Junhao Wang, Maoquan Wang, Yufan Huang, Shengyu Fu, Elsie Nallipogu, Qingwei Lin, Yingnong Dang, Saravan Rajmohan, and Dongmei Zhang. Swe-bench goes live! *arXiv preprint arXiv:2505.23419*, 2025.

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

[48] Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch. *arXiv preprint arXiv:2412.01769*, 2024.

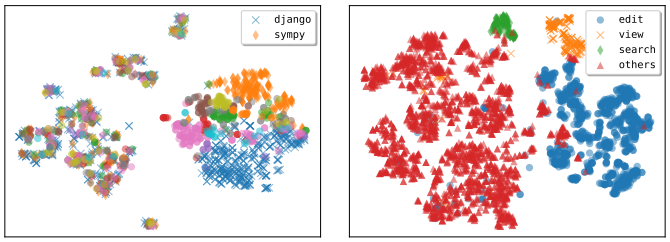
[49] Albert Örwall. Moatless, 2024. `https://github.com/aorwall/moatless-tools`.

A Additional experimental settings

We now describe additional experimental settings used by LIVE-SWE-AGENT for evaluation. As discussed in Section 3, we implement LIVE-SWE-AGENT on top of the mini-SWE-agent framework. We use the same default hyperparameter of mini-SWE-agent at the time of writing (maximum step limit of 250 and maximum cost of $3 per issue).

For Gemini 3 Pro, we use temperature of 1 since it is recommended by the developers[4]. For any Anthropic models in our experiments, we follow mini-SWE-agent and use a temperature of 0.0. For any OpenAI models in our experiments (e.g., GPT-5, GPT-5-Mini, GPT-5-Nano), since temperature 0.0 sampling is not supported, we use temperature of 1. To the best of our knowledge, this is also the case used in mini-SWE-agent when they use OpenAI models.

B Additional tool analysis



(a) Tools in SWE-bench Verified (b) Tools in SWE-Bench Pro

Figure 6: Additional 2 dimensional t-SNE visualization of tools generated by Claude 4.5 Sonnet on SWE-bench Verified and SWE-Bench Pro. We label and display the embedding based on the repository name (Figure 6a) and tool name (Figure 6b). Note that for Figure 6a, we only label two repositories in the legend due to space considerations. The two repositories were chosen as they have representative distinct clusters.

We perform additional tool analysis similar to the ones done in Section 4.2. We added the missing results including the visualization based on repository used for SWE-bench Verified and tool name for SWE-Bench Pro. Note that we did not include any results based on programming language for SWE-bench Verified since they are all repositories in Python. We observe similar patterns to the main evaluation.

To perform this analysis, we use `sklearn` implementation of PCA with `n_components` of 50. We then use the t-SNE implementation from `sklearn` with `max_iter` of 1000.

C Ablation problems

C.1 SWE-bench Verified ablation subset

Here we include the 50 randomly selected problems in SWE-bench Verified used in our ablation evaluations.

- `sympy__sympy-12489`
- `django__django-16502`
- `scikit-learn__scikit-learn-11310`
- `django__django-13590`
- `scikit-learn__scikit-learn-11578`
- `matplotlib__matplotlib-21568`
- `django__django-11276`
- `django__django-11119`
- `sympy__sympy-13031`
- `sphinx-doc__sphinx-8459`
- `django__django-14170`
- `django__django-11066`
- `sphinx-doc__sphinx-9658`
- `django__django-14631`

[4] `https://ai.google.dev/gemini-api/docs/gemini-3`

- django__django-11749
- scikit-learn__scikit-learn-13135
- astropy__astropy-14369
- matplotlib__matplotlib-24177
- django__django-15252
- pydata__xarray-7233
- django__django-11815
- sympy__sympy-18199
- django__django-15467
- psf__requests-5414
- django__django-17084
- scikit-learn__scikit-learn-15100
- django__django-13925
- django__django-11163
- django__django-16595
- django__django-16938
- sympy__sympy-13877
- django__django-16612
- django__django-15629
- pydata__xarray-3677
- django__django-15503
- psf__requests-1142
- mwaskom__seaborn-3069
- django__django-15127
- astropy__astropy-13033
- django__django-16429
- django__django-16256
- django__django-11133
- django__django-13741
- matplotlib__matplotlib-23476
- scikit-learn__scikit-learn-12973
- scikit-learn__scikit-learn-13779
- sympy__sympy-23413
- pytest-dev__pytest-7521
- sympy__sympy-22456
- django__django-16082

C.2 SWE-bench Multilingual ablation subset

We further evaluate on 50 randomly selected subset of problems in SWE-bench Multilingual. Below is the list of `instance_ids` of the problems in our subset.

- apache__druid-13704
- apache__druid-14136
- apache__druid-15402
- apache__lucene-13494
- astral-sh__ruff-15543
- axios__axios-6539
- babel__babel-15445
- burntsushi__ripgrep-2576
- caddyserver__caddy-5404
- caddyserver__caddy-5995
- caddyserver__caddy-6051
- facebook__docusaurus-9183
- faker-ruby__faker-2970
- fastlane__fastlane-20642
- fluent__fluentd-3640
- fmtlib__fmt-3750
- javaparser__javaparser-4538
- javaparser__javaparser-4561
- jqlang__jq-2919
- laravel__framework-48636
- laravel__framework-52451
- mrdoob__three.js-27395
- nushell__nushell-12901
- phpoffice__phpspreadsheet-3659
- preactjs__preact-2757
- preactjs__preact-2927
- preactjs__preact-3562
- preactjs__preact-3739
- projectlombok__lombok-3326
- reactivex__rxjava-7597
- redis__redis-10068
- redis__redis-10095
- rubocop__rubocop-13503
- rubocop__rubocop-13560
- rubocop__rubocop-13627
- rubocop__rubocop-13680
- rubocop__rubocop-13705
- sharkdp__bat-2650
- sharkdp__bat-562
- tokio-rs__axum-1730
- tokio-rs__axum-2096
- tokio-rs__axum-682

- tokio-rs__tokio-6603
- tokio-rs__tokio-6752
- tokio-rs__tokio-6838
- uutils__coreutils-6731
- valkey-io__valkey-1842
- valkey-io__valkey-790
- vuejs__core-11739
- vuejs__core-11870

D Prompts Used

Here we provide the detailed prompts used by LIVE-SWE-AGENT. Note that the majority of the prompt is constructed by mini-SWE-agent and we only make minimal changes to enable on-the-fly tool creation. Furthermore, we perform a slight modification: change the home directory indication of `testbed` to `app` in the initial prompt for all SWE-Bench Pro evaluations. This is done since the SWE-Bench Pro docker environments use `app` as the main source directory containing the repository code, different from SWE-bench environments. Apart from that, we use the same prompt for all our evaluations.

D.1 Initial prompt

Initial prompt for LIVE-SWE-AGENT

```
<pr_description>
Consider the following PR description:
{{task}}
</pr_description>

<instructions>
# Task Instructions

## Overview
You're a software engineer interacting continuously with a computer by submitting commands.
You'll be helping implement necessary changes to meet requirements in the PR description.
Your task is specifically to make changes to non-test files in the current directory in order to fix the issue
↪  described in the PR description in a way that is general and consistent with the codebase.

IMPORTANT: This is an interactive process where you will think and issue ONE command, see its result, then think and
↪  issue your next command.

For each response:
1. Include a THOUGHT section explaining your reasoning and what you're trying to accomplish
2. Provide exactly ONE bash command to execute

## Important Boundaries
- MODIFY: Regular source code files in /testbed (this is the working directory for all your subsequent commands)
- DO NOT MODIFY: Tests, configuration files (pyproject.toml, setup.cfg, etc.)

## Recommended Workflow
1. Analyze the codebase by finding and reading relevant files
2. Create a script to reproduce the issue
3. Edit the source code to resolve the issue
4. Verify your fix works by running your script again
5. Test edge cases to ensure your fix is robust

## Command Execution Rules
You are operating in an environment where
1. You write a single command
2. The system executes that command in a subshell
3. You see the result
4. You write your next command

Each response should include:
1. A **THOUGHT** section where you explain your reasoning and plan
2. A single bash code block with your command

Format your responses like this:

<format_example>
THOUGHT: Here I explain my reasoning process, analysis of the current situation,
and what I'm trying to accomplish with the command below.

```bash
your_command_here
```
</format_example>

Commands must be specified in a single bash code block:

```bash
your_command_here
```
```

````
**CRITICAL REQUIREMENTS:**
- Your response SHOULD include a THOUGHT section explaining your reasoning
- Your response MUST include EXACTLY ONE bash code block
- This bash block MUST contain EXACTLY ONE command (or a set of commands connected with && or ||)
- If you include zero or multiple bash blocks, or no command at all, YOUR RESPONSE WILL FAIL
- Do NOT try to run multiple independent commands in separate blocks in one response
- Directory or environment variable changes are not persistent. Every action is executed in a new subshell.
- However, you can prefix any action with `MY_ENV_VAR=MY_VALUE cd /path/to/working/dir && ...` or write/load environment variables from files

Example of a CORRECT response:
<example_response>
THOUGHT: I need to understand the structure of the repository first. Let me check what files are in the current directory to get a better understanding of the codebase.

```bash
ls -la
```
</example_response>

Example of an INCORRECT response:
<example_response>
THOUGHT: I need to examine the codebase and then look at a specific file. I'll run multiple commands to do this.

```bash
ls -la
```

Now I'll read the file:

```bash
cat file.txt
```
</example_response>

If you need to run multiple commands, either:
1. Combine them in one block using && or ||
```bash
command1 && command2 || echo "Error occurred"
```

2. Wait for the first command to complete, see its output, then issue the next command in your following response.

## Environment Details
- You have a full Linux shell environment
- Always use non-interactive flags (-y, -f) for commands
- Avoid interactive tools like vi, nano, or any that require user input
- If a command isn't available, you can install it

## Useful Command Examples

### Create a new file:
```bash
cat <<'EOF' > newfile.py
import numpy as np
hello = "world"
print(hello)
EOF
```

### View file content:
```bash
# View specific lines with numbers
nl -ba filename.py | sed -n '10,20p'
```

**IMPORTANT TOOL CREATION INSTRUCTIONS**
## Creating your own tools
- You can also create your own tools in Python to help with your workflow
- Compared to basic bash commands, the tools you create should be able to better aid your workflow in solving the task
- Ensure each tool you create is in Python, contains informative outputs or error messages, and can be ran from the command line
- You should at least create a simple edit tool that can help you effectively edit arbitrary files instead of using bash commands
- The tools you create can be for any purpose, it does not need to be general, instead think about how it can help you specifically with the current task at hand

### Example of creating a custom tool:
<example_response>
THOUGHT: I noticed that in order to solve the issue I need to ... therefore I should create a custom tool to help me ...

```bash
cat <<'EOF' > /path/to/tool_name.py
#!/usr/bin/env python3
import sys
# Import other packages if needed

def main():
    # Your tool logic here
    ...
````
````

````
if __name__ == "__main__":
    main()
EOF
```
</example_response>

### Example of using the tool you created:
<example_response>
THOUGHT: Let me use the custom tool I created to help me with ...

```bash
python /path/to/tool_name.py <<EOF
your_input_here
EOF
```
</example_response>

## Submission
When you've completed your work (reading, editing, testing), and cannot make further progress
issue exactly the following command:

```bash
echo COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT && git add -A && git diff --cached
```

This command will submit your work.
You cannot continue working (reading, editing, testing) in any way on this task after submitting.
</instructions>
````

Figure 7: The initial prompt for LIVE-SWE-AGENT, modified from the base mini-SWE-agent prompt.

D.2 Feedback message

Feedback message used by LIVE-SWE-AGENT

```
<returncode>{{output.returncode}}</returncode>
{% if output.output | length < 10000 -%}
<output>
{{ output.output -}}
</output>
Reflect on the previous trajectories and decide if there are any tools you can create to help you with the current
↪  task.
Note that just because you can use basic bash commands doesn't mean you should not create any tools that can still be
↪  helpful.
{%- else -%}
<warning>
The output of your last command was too long.
Please try a different command that produces less output.
If you're looking at a file you can try use head, tail, sed or create a tool to view a smaller number of lines
↪  selectively.
If you're using grep or find and it produced too much output, you can use a more selective search pattern.
</warning>
{%- set elided_chars = output.output | length - 10000 -%}
<output_head>
{{ output.output[:5000] }}
</output_head>
<elided_chars>
{{ elided_chars }} characters elided
</elided_chars>
<output_tail>
{{ output.output[-5000:] }}
</output_tail>
{%- endif -%}
```

Figure 8: The feedback message used by LIVE-SWE-AGENT after each agent step.
```