PROGRAMMING WITH PIXELS: TOWARDS GENERALIST SOFTWARE ENGINEERING AGENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advancements in software engineering (SWE) agents have largely followed a *tool-based paradigm*, where agents interact with hand-engineered tool APIs to perform specific tasks. While effective for specialized tasks, these methods fundamentally lack generalization, as they require predefined tools for each task and do not scale across programming languages and domains. We introduce `Programming with Pixels` (`PwP`), an agent environment that unifies software development tasks by enabling *computer-use agents*—agents that operate directly within an IDE through visual perception, typing, and clicking, rather than relying on predefined tool APIs. To systematically evaluate these agents, we propose `PwP-Bench`, a benchmark which unifies existing SWE benchmarks spanning tasks across multiple programming languages, modalities, domains under a task-agnostic state and action space. Our experiments demonstrate that general-purpose computer-use agents can approach or even surpass specialized tool-based agents on a variety of SWE tasks without the need for hand-engineered tools. However, our analysis shows that current models suffer from limited visual grounding and fail to exploit many IDE tools that could simplify their tasks. When agents can directly access IDE tools, without visual interaction, they show significant performance improvements, highlighting the untapped potential of leveraging built-in IDE capabilities. Our results establish `PwP` as a scalable testbed for building and evaluating future computer-use SWE agents that interact directly with development environments.

1 INTRODUCTION

Human software developers possess a remarkable ability to work across a wide range of programming tasks, seamlessly adapting to new languages, tools, and problem domains. Realizing a single, general-purpose agent with similar versatility is the overarching goal of many recent efforts in code generation and software engineering automation Jiang et al. (2024); Jin et al. (2024); Wang et al. (2024b). However, most software engineering agents (SWE agents) still rely on a *tool-based paradigm*, where an agent takes actions using hand-engineered functions (e.g., search repository, run Python code) exposed through a text API Yang et al. (2024a;b); Wang et al. (2024a;b). This fundamentally limits generalization, since tool-based agents can only perform tasks using the predefined actions. For example, an agent designed to manage GitHub pull requests lacks debugging abilities unless it is programmed into the agent's API. Furthermore, the tool-based paradigm lacks scalability, as hand-engineering complex tools requires significant human effort and may not be bug-free. As a result, it remains unclear whether the tool calling paradigm scales to the diversity of software engineering tasks, which spans multiple languages, modalities, and task types.

Our motivating hypothesis is that achieving general-purpose SWE agents requires a shift to *computer-use agents* Anthropic (2024) that interact with computers as humans do: by observing the screen, typing, and clicking. To this end, we recast agentic software engineering as interacting directly with an *integrated development environment (IDE)* by observing its visual state and using basic actions such as clicking and typing. This allows the agent to perform any task possible in an IDE, and leverage all of the IDE's tools—from debuggers to web browsers—without requiring specialized APIs. However, despite promising results in web navigation Anthropic (2024) and open-ended computer tasks Xie et al. (2024), we lack a dedicated environment for software engineering, and the ability of computer-use agents to perform software engineering remains underexplored.

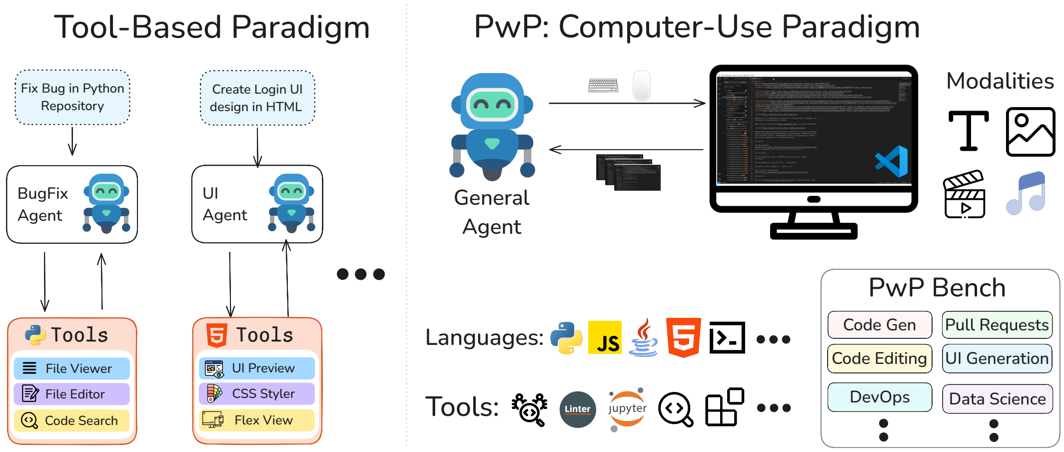

Figure 1: Comparison between traditional tool-based paradigm (left) and our proposed `Programming with Pixels` (`PwP`) framework (right) for software engineering (SWE) agents. Instead of relying on specialized hand-engineered tools, `PwP` enables agents to interact directly with an IDE through basic computer interactions and screen observation. The framework naturally integrates with existing IDE capabilities across multiple languages and tools. Further `PwP-Bench` is a comprehensive benchmark to evaluate agent performance across different SWE domains.

To close this gap, we introduce *Programming with Pixels (`PwP`)*, the first software engineering agent environment aimed at general-purpose computer-use agents. The `PwP` environment is a VSCode-based IDE where agents perceive the screen and use primitive actions such as typing, pointing, and clicking. `PwP` fulfills two key properties. First, the environment is *expressive*, allowing agents to complete any software engineering task achievable in an IDE, without language- or domain- specific modifications. Second, agents naturally interact with *any tools available in the IDE*–including debuggers, linters, and code suggestions while handling diverse data types such as images, videos, PDFs–through basic actions such as clicking and typing. This notion of tool use fundamentally differs from hand-engineered tool APIs, offering scalability and reducing hand-engineering effort for AI agents. Namely, `PwP` lets agents take advantage of the rich tools already accessible to humans in an IDE, rather than reinventing the wheel. Finally, computer-use agents reduce complex tool pipelines (e.g., tool-specific prompts), opening up a simplified approach to general-purpose SWE agents.

To further evaluate agents developed for computer-use we construct `PwP-Bench`, a unified benchmark of 15 tasks spanning a variety of software engineering activities, including code generation, pull request resolution, UI development, and DevOps workflows. We show for the first time that general-purpose computer-agents can achieve non-trivial performance on a wide variety of tasks, often approaching or even exeeding state-of-the-art tool-based agents. However, our analysis reveals substantial opportunities for future work. First, even state-of-the-art computer-use agents suffer from visual grounding issues. Second, we show that current agents lack the ability to use many of the tools available in the IDE, including ones that could make their tasks trivial. This suggests that training agents to explore and use the tools available in the IDE is a fruitful future direction. Finally, we find that only one model (Claude (Anthropic, 2024)) performs well, highlighting the need for further research into training and improving computer-use agents.

In summary, our contributions are five-fold: First, we introduce `Programming with Pixels` (`PwP`), the first software engineering-focused environment for evaluating computer-use agents. Second, we propose `PwP-Bench`, a benchmark spanning 15 diverse SWE domains, allowing for systematic comparison of software engineering agents. Third, we demonstrate, for the first time, that computer-use agents can perform a wide variety of software engineering tasks without additional hand-engineering of their action or observation space. Fourth, we analyze the limitations of current computer-use agents, identifying the need for models that better leverage IDE tooling and highlighting agent training as a key future direction. Finally, both existing agents and benchmarks can be easily incorporated into our unified environment, positioning `PwP` to serve as a common platform for developing future SWE agents. Overall, `PwP` challenges the prevailing tool-based paradigm for SWE agents and provides a platform for developing more general agents that interact directly with IDEs.

2 RELATED WORK

Task-specific SWE benchmarks Prior work has focused on isolated tasks like code generation from docstrings Chen et al. (2021), pull request resolution Jimenez et al. (2023), and multimodal programming Si et al. (2024). While valuable, these benchmarks are confined to specific languages, modalities, or task types. In contrast, `PwP-Bench` unifies these diverse evaluations into a single framework, encompassing multimodal and multilingual challenges that require extensive interaction with IDE tools.

SWE Agents Recent code agents have moved toward interactive approaches but often specialize in particular tools or languages Jin et al. (2024); Xia et al. (2024). For instance, Agentless relies on Python-specific tools Xia et al. (2024), while SWE-agent requires task-specific modifications Yang et al. (2024a) (See Appendix A.1 for comparison with other methods). In contrast, `PwP` agents are inherently task and language-agnostic due to our environment's expressive action space, with tools available directly within the IDE rather than requiring hand-engineered solutions.

Visual Agents and Computer-Use Agents Visual agents Koh et al. (2024a); Deng et al. (2023) typically rely on limited action sets and struggle with complex IDE interfaces. While computer-use agents Anthropic (2024); OpenAI (2025) offer more expressive interactions, they lack SWE-specific environments for evaluation. `PwP` bridges this gap by providing a unified IDE platform for testing and developing such agents on realistic SWE tasks.

Expressive Agent Environments Existing environments like OSWorld Xie et al. (2024) target general scenarios but lack SWE focus. `PwP` specifically addresses software engineering challenges while maintaining compatibility with existing agents through IDE modifications that enable direct tool access and state information.

For detailed comparisons and extended discussion, we urge readers to check Appendix B.

3 PROGRAMMING WITH PIXELS (PWP)

Modern software engineering (SWE) often requires using multiple programming languages, tools, and modalities. Furthermore, it relies on a wealth of tools that required tremendous human effort to create: from linters, to visual code debuggers, to project management tools. Motivated by these observations, we create Programming with Pixels (PwP), an IDE environment that satisfies two properties: (i) it is *expressive*, meaning that an agent can perform any task that is achievable through a sequence of primitive operations (e.g., typing or clicking) within an IDE; (ii) an agent has access to any tool implemented within the IDE, since using a tool amounts to performing a sequence of primitive actions.

3.1 PWP ENVIRONMENT

We represent the `PwP` environment as a partially observable Markov decision process (POMDP). We define the PwP POMDP $\langle S, A, O, T, R \rangle$ as follows. The state space $\mathbf{S}$ describes the IDE and operating system context, including open files, active editor panels, and cursor positions. The action space $\mathbf{A}$ encompasses all possible keyboard and mouse events, with atomic actions provided by the `xdotool` library Sissel in a simple syntax. Both $\mathbf{A}$ and the observation space $\mathbf{O}$ further varies based on the agent setting (§5). The transition function $\mathbf{T}$ handles both deterministic changes (e.g., character insertion) and stochastic elements from background processes. Finally, the reward function $\mathbf{R}$ measures task performance, for instance by running test suites on updated files after bug fixes.

Trajectories in `PwP` can thus resemble real-world development work: an agent can fix a bug in a repository, use a suggestion tool to help with writing code, or create documentation. The IDE and its operating system environment track changes, run tests, and return reward signals.

3.2 KEY FEATURES OF PROGRAMMING WITH PIXELS

Expressive observation and action space. A typical approach to building agents is to engineer at set of high-level actions for operations like "open file" or "list symbols in file", and then engineer an environment that supports each action Xia et al. (2024); Yang et al. (2024b); Wang et al. (2024b).

Table 1: **Comparison of PwP with existing environments** across different dimensions. PwP uniquely combines comprehensive IDE tool support with full multimodal support, general action space, and execution-based evaluation, while maintaining software engineering specificity.

Environment	Multi-modal	General Action Space	Observation Space	State Checkpointing	Tools	Execution-Based Reward	SWE Specific
GAIA Mialon et al. (2023)	✗	✗	Text	✗	Limited	✗	✗
SWE-Bench Jimenez et al. (2023)	✗	✗	Text	✗	Limited	✓	✓
SWE-Bench-MM Yang et al. (2024b)	✓	✗	Text	✗	Limited	✓	✓
WEBSHOP Yao et al. (2023)	✗	✗	Text	✗	Browser	✗	✗
WEBARENA Zhou et al. (2024)	✗	✗	Text	✗	Browser	✗	✗
VWEBARENA Koh et al. (2024a)	✓	✓	Screen	✗	Browser	✗	✗
OpenHands Wang et al. (2024b)	Text, Image	✗	Tool Output	✗	SWE	✓	✓
TheAgentCompany Xu et al. (2024)	Text, Image	✗	Tool Output	✗	SWE	✓	✓
OSWORLD Xie et al. (2024)	Text, Image	✓	Screen	✗	OS	✓	✗
WindowsAgentArena Bonatti et al. (2024)	Text, Image	✓	Screen	✗	OS	✓	✗
PwP (Ours)	Text, Image, Video, Audio	✓	Screen	✓	All IDE Tools	✓	✓

Furthermore, agents receive textual outputs that are manually reformatted for each action. The key difficulty is that such engineering does not scale to a large number of actions or to the full range of software engineering tasks, and the agent may be specialized to the observation and action space. In contrast, PwP preserves the standard screen-based user interaction. The agent can navigate IDE menus visually, moves the cursor, and presses keys. This makes the environment expressive: an agent can achieve any task that can be achieved through a sequence of primitive actions in an IDE.

Full Spectrum of Developer Tools. A modern IDE offers debuggers, linters, version control, refactoring utilities, integrated terminals, and many extensions. In particular, PwP is developed on top of VSCode, which has a rich set of built-in functionalities and extensions. Implementing each of these into an agent or environment would require significant human effort. However, PwP provides these capabilities out-of-the-box within a single environment. Agents can set breakpoints, execute code, use language-specific extensions, review error messages, or run tests in a consistent interface.

Multimodality and Language Agnosticism. Because the IDE supports numerous programming languages through extensions and built-in modules, PwP naturally covers tasks across Python, Java, JavaScript, Lean and more without requiring separate integrations. For instance, agents can use the same debugger interface for all languages, or use pre-existing linters provided by IDE extensions. Beyond screenshots, the environment provides video streams and audio, though we leave exploring these for future work.

Rich Feedback and State Access. PwP can evaluate performance immediately using testing frameworks or compilation checks. For instance, if the agent modifies a file, the IDE can automatically trigger a build, update diagnostics, or run tests, generating real-time feedback. In cases requiring deeper inspection—such as verifying that a bug is truly fixed—the environment can reveal file-system changes, process states, or test results.

Future adaptability. As agents continue to evolve, PwP provides a unified environment for incorporating new benchmarks and training. For example, PwP is amenable to reinforcement learning due to its use of the standard gymnasium Towers et al. (2024) interface and its checkpointing functionality. We show an example of how to interact with PwP in Figure 5 Checkpointing also allows for backtracking in search-based methods Koh et al. (2024b); Putta et al. (2024). As software-engineering practices progress and new IDE tools emerge, PwP incorporates them without additional engineering overhead. Further, as agents become more ubiquitous, it is imperative to evaluate their capabilities in pair-programming scenarios with human developers. PwP also supports concurrent user-agent interaction, potentially enabling new kinds of pair-programming or real-time collaboration studies. Finally, adding new tasks requires only minimal modification to configuration and evaluation files.

3.3 Infrastructure and Implementation

PwP is deployed in a secure sandboxed environment. In particular, we run a modified version of Visual Studio Code (VSCode) and a minimal operating system inside a Docker container, ensuring a secure and isolated environment. We chose VSCode for its extensive language support, rich ecosystem of extensions, widespread adoption in the developer community, and open-source nature that enables customization and modification of its core functionality. Each container instance maintains its own file system and processes, preventing interference between experiments, facilitates reproducibility, and ensures parallelization of evaluation. We further provide the ability to checkpoint environment state, especially useful for backtracking while training RL agents.

The environment interfaces with VSCode through multiple channels: 1.) A controller that manages Docker container lifecycle and configuration, 2.) A port-forwarding system for real-time screen and video capture, 3.) A modified VSCode codebase that exposes DOM state information, and 4) The VSCode Extension API for accessing fine-grained IDE state. This multi-channel approach enables both high-level environment control and detailed state observation.

Screen capture is handled via `ImageMagick` for static screenshots and `ffmpeg` for streaming video output. These tools were selected for their low latency and ability to handle various screen resolutions and color depths. For actions, a lightweight controller executes `xdotool` commands within the container, which in turn simulates keyboard and mouse events on the IDE. Agents can thus insert code, open new files, or navigate menus using the same actions that a human developer would.

A Python API is provided for interaction, following a style similar to common reinforcement learning libraries such as gymnasium Towers et al. (2024). The API abstracts away container management complexity, handling observations and actions, allowing researchers to focus on agent development. Users can query the environment for the latest screenshot, issue an `xdotool` command, and receive updated states or rewards. The environment's container configuration is flexible, allowing arbitrary software installations, customizable CPU/memory limits, and display settings. This versatility is crucial for large-scale evaluation, especially when tasks vary in complexity and resource needs.

4 PWP-BENCH

We introduce `PwP-Bench`, a benchmark comprising 15 diverse software engineering tasks that span 8 programming languages and multiple modalities. Each task provides agents with access to the complete suite of tools available in the `PwP` environment. The purpose of `PwP-Bench` is to assess how well agents handle a broad range of software engineering (SWE) activities, thereby testing the generality of their code-generation and SWE capabilities.

Tasks `PwP-Bench` contains 5400 instances covering 15 tasks, sourced from 13 existing code-generation datasets and 2 newly created by us. These tasks are designed to be representative of the breadth of software engineering—including tasks beyond conventional code generation to capture real-world complexities—and can be expanded as models excel in newer tasks. We followed three guiding principles: (1) tasks should require significant interaction with various SWE tools, (2) each task should necessitate multiple steps to complete, and (3) the overall benchmark should span multiple programming languages and modalities. Based on these principles, we collected diverse tasks and grouped them into four categories:

- **Code Generation and Editing:** Evaluates the ability of agents to generate and edit code. This category includes datasets such as HumanEval for code completion, SWE-Bench Jimenez et al. (2023) and SWE-Bench-Java Zan et al. (2024) for resolving pull requests, DSBench Jing et al. (2024) for data science tasks, and Res-Q LaBash et al. (2024) or CanITEdit Cassano et al. (2024) for code editing. Each dataset benefits from different tools; for example, SWE-Bench can take advantage of debuggers and linters, whereas DSBench may leverage an IPython kernel and tools for analyzing large data files. Code editing tasks further require refactoring utilities and repository searches, covering varied input-output formats and end goals.
- **Multimodal Code Synthesis:** Involves creating code based on input images or other visual data. Examples include Design2Code Si et al. (2024) for UI development, Chart2Mimic Shi et al. (2024) for generating Python code from chart images, SWE-Bench-MM Yang et al. (2024b) for multimodal code editing, and DSBench tasks that rely on images or PDF documents during data analysis.
- **Domain-Specific Programming:** Focuses on specialized fields such as ethical hacking (CTF) Yang et al. (2023b) and theorem proving (miniCTX) Hu et al. (2024). These tasks demand significant interactivity with IDE components. For example, theorem proving requires continuous inspection of goal states via an interactive interface, while CTF tasks often involve analyzing images, running executables, or installing VSCode extensions (e.g., hexcode readers).
- **IDE-Specific and General SWE Tasks:** Recognizing that code generation is only one aspect of software engineering, we introduce two novel task sets to evaluate broader SWE skills. The first,

IDE Configuration, assesses an agent's ability to modify IDE settings—such as themes, extension installations, and preferences—that are critical for effective tool use in a complex environment. The second, which we term **General SWE**, targets non-code activities such as profiling, designing UI mockups, managing Kanban boards, and project refactoring. These tasks capture essential operational skills typically required by human developers but largely absent from conventional code generation benchmarks.

Note that a single task may appear in more than one category. Figure 2 shows the distribution of tasks across all categories. In total, `PwP-Bench` covers Python, Java, JavaScript, HTML, CSS, Bash, SQL, and Lean, requiring agents to work with text, images, data files, and other data types. Effective interaction with IDE tools is essential; an agent that succeeds across these tasks demonstrates strong potential for automating a wide range of software engineering activities.

Benchmarking Design and Task Setup Every task is evaluated within the `PwP` environment. Unlike traditional benchmarks that provide structured, well-formatted context (for example, supplying all relevant schemas in text-to-SQL), `PwP-Bench` presents agents with an IDE containing a codebase rich in information. Specifically, an agent is provided with an initial environment state S_i and an instruction I. The agent's goal is to update the codebase to satisfy I and transition to a final state S_f. Only S_f is evaluated, using execution-based criteria (e.g., running unit tests). This setup requires agents to autonomously discover and extract relevant information from files, directories, and other resources—mirroring the challenges faced in real-world software development.

Many tasks in `PwP-Bench` require extensive multi-turn interactions and can be time-consuming. To support large-scale evaluations, `PwP` enables parallelized testing in a sandboxed environment, ensuring both security and reproducibility. Tasks can also be configured to restrict or partially allow internet access based on experimental needs. Furthermore, as software engineering tasks evolve with advancements in model capabilities, our benchmark is designed to grow over time. New tasks can be incorporated by creating simple setup scripts that define the IDE's initial state and evaluation logic, ensuring that `PwP-Bench` remains modular and adaptable.

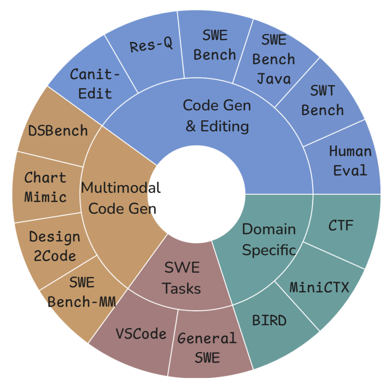

Figure 2: Distribution of tasks in PWP-Bench across four main categories: Code Generation and Editing, Multimodal Code Synthesis, Domain-Specific Programming, and General SWE Tasks. The inner ring shows the main categories while the outer ring shows datasets and tasks within each category.

`PwP`-Lite Because `PwP-Bench` contains more than 5400 instances in total, running a full evaluation can be computationally expensive. To address this, we also provide `PwP-Bench`-Lite—a smaller subset comprising 300 instances, with 20 random samples per task. This subset preserves the overall difficulty and distribution while ensuring equal representation for each task, thereby making rapid experimentation more accessible.

5 Agents in Programming with Pixels

The primary objective of Programming with Pixels is to enable general-purpose SWE agents. We evaluate state-of-the-art agents based on vision-language models in our environment. Each agent operates in a turn-based manner, receiving a screenshot each turn and returning an action (keyboard or mouse action) to progress toward the goal. This design is same as used in previous works Xie et al. (2024); Koh et al. (2024a) and we refer them to as *computer-use agents*. In practice, most vision-language models struggle with raw image inputs. To mitigate this, we incorporate *Set-of-Marks (SoM)* Yang et al. (2023a), in which the agent receives both the raw image and a parse of available interface element (e.g., buttons, text fields). The agent can then interact with the desired element ID instead of raw pixel coordinates.

Table 2: Performance Evaluation of Different Agents on PwP-Bench by Task Categories

Inputs	Outputs	Model	Code Generation & Editing	Multimodal Code Generation	Domain-Specific Code Generation	General SWE Tasks	Overall Avg
Screenshot + SoM	Keyboard + Mouse	GPT-4o	0.8%	16.4%	1.7%	10.0%	6.4%
		GPT-4o-mini	0.8%	8.0%	0.0%	2.5%	2.8%
		Gemini-Flash	0.0%	10.6%	0.0%	0.0%	2.8%
		Gemini-Pro	2.5%	11.9%	0.0%	7.5%	5.2%
		Claude-Sonnet	10.0%	18.5%	5%	**20.0%**	13.0%
Screenshot + SoM + Tool Output	Keyboard + Mouse + Tool Call (File, Bash)	GPT-4o	37.0%	48.1%	28.3%	5.0%	34.0%
		GPT-4o-mini	23.6%	25.4%	15.0%	5.0%	19.9%
		Gemini-Flash	10.4%	19.9%	8.3%	2.5%	11.1%
		Gemini-Pro	30.9%	24.7%	3.3%	5.0%	20.3%
		Claude-Sonnet	**52.1%**	**57.7%**	43.3%	**20.0%**	**47.2%**
Previously Reported State of the Art			53.7%	54.6%	**51.2%**	-	-

We evaluate two categories of computer-use agents. The first category only outputs keyboard and mouse clicks. In summary, $\mathbf{O}$ = a screenshot and set-of-marks annotations. $\mathbf{A}$ = keyboard and mouse clicks with set-of-marks.

The second category of computer-use agents has access to file and bash commands supplied by the environment through an API. These file and bash actions are provided in PwP in similar design principle as Anthropic computer-use Anthropic (2024) which consists of file operations such as read file, create file, and string replace. In summary, $\mathbf{O}$ = a screenshot and set-of-marks annotations and text output from actions if performed. $\mathbf{A}$ = keyboard, mouse, and file and bash operation actions.

While the above agent design favors generalizability and simplicity, we note that current vision-language models, and in particular prior works on tool-based agents, are incapable of interacting with computers using primitive observation and action spaces. To support such agents, in Analysis 6.2, we create a tool-based agent design compatible with PwP. We provide agents with domain-specific tools, enabling them to perform high-level actions (such as getting file structure) through API calls instead of UI interactions. Each high-level action is implemented as a sequence of low-level actions executed in PwP. This setting lets us test current state-of-the-art SWE agents within the PwP environment.

6 Experiments

Experimental setup We evaluate two categories of baseline agents as described in Section 5. In each configuration, we employ five state-of-the-art vision-language models: Gemini-Flash-1.5, Gemini-Pro-1.5, GPT-4o, GPT-4o-mini, and Claude-3.5 Sonnet. With the exception of Claude—which is natively trained for UI interaction—the remaining models are provided with a SoM image.

At each timestep, an agent receives an observation (with the observation space determined by its category) and returns an action. The complete history of observations and actions is incorporated into the model's context. For each task instance, the maximum number of iterations is capped at 20 steps; if the agent either exhausts these steps or issues a stop command, the environment's final state is evaluated using task-specific metrics (see Appendix C for full details). Notably, the agent design remains the same throughout all tasks. Due to computational and budget constraints, we evaluate on PwP-Bench-Lite, which comprises 300 task instances.

6.1 Results

Table 2 summarizes performance across different agent architectures and base models over the four categories of PwP-Bench. When only screenshots are provided to the VLMs, they achieve near-zero accuracy in most categories, with a maximum overall average of 13.0%. We attribute this poor performance primarily to limited visual grounding and an inability to interact effectively with the IDE—particularly for file editing and tool usage (see Section 6.2 for further analysis).

In contrast, when agents are granted access to file editing and bash operations through API calls rather than relying solely on UI interactions, we observe consistent improvements across all categories, with maximum average accuracy reaching 47.2%. Among the evaluated models, the Claude computer-use agent performs best, likely because it is specifically trained for UI interactions. As a result,

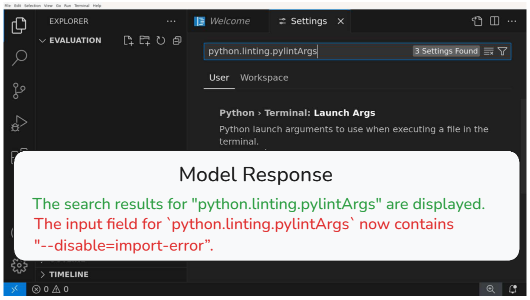

Figure 3: **Example of Agent Hallucinating Screen Contents** The agent hallucinates an input field containing "disable import error" (red)

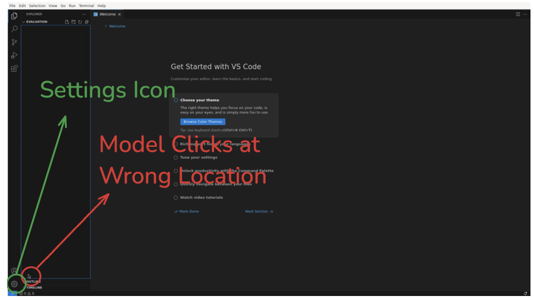

Figure 4: **Wrong mouse click by Claude-Computer Use Agent** The agent attempted to click Settings icon but clicked at wrong position.

this agent leverages basic IDE tools—such as HTML live preview, chart visualization, and file navigation—to boost performance on tasks requiring visual understanding and IDE navigation. Notably, we demonstrate for the first time that a single computer-use agent can achieve performance comparable to state-of-the-art methods across a wide variety of SWE tasks—encompassing multiple languages, modalities, and domains—while operating within a unified environment and interface.

Nonetheless, as detailed in Section 6.2, the models are currently incapable of using the tooling available in the IDE. If they could use the IDE more effectively, performance would likely improve further across the board. This is evidenced by the poor performance on the 'General SWE' dataset, where tasks are often as simple as editing IDE settings and often require fewer than four clicks to complete. As we show in Section 6.2, these tasks become simpler if the models could use the IDE tooling more effectively. Overall, while the results point toward a promising direction for developing general computer-use SWE agents, significant improvements are still needed in visual grounding, tool usage, and planning capabilities. We analyze these in the following sections.

6.2 Analysis

Agents Demonstrate Poor Visual Grounding Capabilities Our qualitative analysis across multiple VLMs on `PwP-Bench` reveals significant limitations in visual understanding—even for basic IDE interactions. We identify two primary failure modes. First, models frequently fail to correctly identify the UI elements intended for interaction, as demonstrated in Figure 3, 4. In agents using SOM, this issue manifests as incorrect element selection, while without SoM it leads to inaccurate mouse positioning. Second, models struggle to comprehend the current UI state. As shown in Figure 8, 9, they consistently fail to recognize highlighted elements, cannot detect linter errors indicated by wavy underlines (Figure 6), and often confuse active panels—resulting, for example, in typing into search bars rather than file editors (Figure 8). While similar issues have been documented in web and OS domains Koh et al. (2024a); Xie et al. (2024), these limitations were primarily observed in models without UI-specific training. However, our work, shows even models explicitly trained for UI interaction Anthropic (2024), including Claude-Computer Use, exhibit these issues in `PwP`—likely due to the increased complexity of the IDE interface.

Agents Struggle with File Editing and Error Recovery. Our analysis reveals significant limitations in agents' file editing capabilities, even for models specifically trained on UI interactions. These models commit basic editing errors and struggle with tasks like indentation—likely due to either overfitting to simpler interfaces or the increased complexity of IDE environments (See Figure 6, 7). While direct file access is available, this limitation prevents agents from leveraging valuable IDE features like notebook editing and visual diff comparisons. Furthermore, agents demonstrate poor error recovery capabilities, often persisting with failed actions or incorrect solution paths. When we deliberately suppressed actions, agents continued their planned sequences despite clear visual feedback showing unchanged states, suggesting a concerning reliance on memorized action sequences rather than dynamic environmental adaptation. We refer readers to Appendix D.1 for more examples.

Agents struggle at using IDE functionality. We find that computer-use agents perform poorly when utilizing the tools provided within the IDE. For instance, we observed no instances of using debuggers or listing symbols in code files. Models that are not specifically trained for UI interaction struggle with even basic tools such as HTML live preview, previewing images in the codebase, and generating or visualizing images and graphs. Among the evaluated models, only Claude demonstrates the ability to use these basic tools, as evidenced by performance improvements observed across multimodal tasks. However, even Claude is limited to the simplest tool functions; it is unable to use more advanced tools, such as profilers or debuggers. To evaluate these capabilities in detail, we constructed 'General-SWE' dataset—where the objective is to perform software engineering activities (e.g., profiling, refactoring, debugging) without editing or writing code. Although these tasks can typically be completed in 4-5 steps using IDE tools, the agents achieve only trivial performance, highlighting the potential for improvement in tool usage.

Training models to use IDE tools better would improve performance. While a single computer-use agent design can perform well across a wide variety of tasks, our results indicate that these models do not fully exploit domain-specific tools. As an indication of the potential for performance gains *if* the agent was able to effectively use the IDE, we perform an "assisted" experiment.

Table 3: Comparison of Different Agent Types Across Selected Tasks

	SWE-Bench	Design2Code	Chartmimic	BIRD
Computer Use I	0%	23.5%	2.7%	0%
Computer Use II	15%	48.1%	25.3%	7%
Assisted	**19%**	**79.5%**	**61.6%**	**17%**

In this experiment, we manually engineer a set of API calls that are useful for the tasks. For example, in Design2Code, assisted agents has a API call for a live HTML preview, while for SweBench it has API calls for retrieving repository structure and symbol outlines. Importantly, each API call is achievable using basic operations in the IDE, meaning that in principle an agent could learn to perform it. To ensure that each API call is achievable in the IDE, we implement each API call by executing a fixed sequence of low-level IDE actions, with the details abstracted away from the agent.

Table 3 compares the performance of these assisted agents with that of standard computer-use agents across four datasets for which we manually created tools. The assisted agents achieve up to a 13.3% improvement in average scores relative to the non-assisted agents. This suggests that training agents to explore and use the built-in IDE functionality would yield performance gains. It also suggests that in the near term, we can get performance gains by introducing hand-engineered tools into the computer-use agent and incorporate existing agent designs in our unified `PwP` environment.

Further, our 'General-SWE' tasks specifically evaluate scenarios where IDE tool usage would be beneficial. In one representative example involving symbol renaming across a project, Claude achieves 0% accuracy when attempting the task without tools. When explicitly instructed to use the renaming tool, its accuracy improves to 50% (See Appendix D). However, this improvement is limited to simple tools—when presented with more complex tools like debuggers, the agent fails to utilize them entirely. These results further emphasize the potential for improving tool-use through computer-interaction for improving performance.

7 CONCLUSION

In this work, we introduce `PwP`, a unified environment that challenges the prevailing tool-based paradigm by enabling direct interaction with IDEs through basic computer-use actions like typing and clicking. This approach allows for a wide range of tasks to be modeled without language- or domain-specific modifications, demonstrated through `PwP-Bench`, a unification of existing SWE datasets evaluated consistently in `PwP`. Our results show that general-purpose computer-agents can approach or outperform previous state-of-the-art results, without any task-specific improvements. This suggests that the dominant paradigm of building specialized text-based tools for SWE agents may be superseded by end-to-end computer-use agents. However, our analysis reveals even state-of-the-art agents are still incapable of using the extensive set of tools available in `PwP`, and could perform better if they could use them. Our work opens up an exciting new direction of development of computer-use agents for SWE tasks, an important step towards reaching truly general purpose SWE agents. By providing a common platform that can incorporate both existing agents and benchmarks, `PwP` positions itself as a foundation for future research in this direction.

REFERENCES

Talor Abramovich, Meet Udeshi, Minghao Shao, Kilian Lieret, Haoran Xi, Kimberly Milner, Sofija Jancheska, John Yang, Carlos E. Jimenez, Farshad Khorrami, Prashanth Krishnamurthy, Brendan Dolan-Gavitt, Muhammad Shafique, Karthik Narasimhan, Ramesh Karri, and Ofir Press. Enigma: Enhanced interactive generative model agent for ctf challenges, 2024. URL https://arxiv.org/abs/2409.16165.

Aider. o1 tops aider's new polyglot leaderboard. https://aider.chat/2024/12/21/polyglot.html, 2024. Accessed: 2025-02-12.

Anthropic. Developing a computer use model, October 2024. URL https://www.anthropic.com/news/developing-computer-use.

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models, 2021. URL https://arxiv.org/abs/2108.07732.

Rogerio Bonatti, Dan Zhao, Francesco Bonacci, Dillon Dupont, Sara Abdali, Yinheng Li, Yadong Lu, Justin Wagle, Kazuhito Koishida, Arthur Bucker, Lawrence Jang, and Zack Hui. Windows agent arena: Evaluating multi-modal os agents at scale, 2024. URL https://arxiv.org/abs/2409.08264.

Federico Cassano, Luisa Li, Akul Sethi, Noah Shinn, Abby Brennan-Jones, Jacob Ginesin, Edward Berman, George Chakhnashvili, Anton Lozhkov, Carolyn Jane Anderson, and Arjun Guha. Can it edit? evaluating the ability of large language models to follow code editing instructions, 2024. URL https://arxiv.org/abs/2312.12450.

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, 2021. URL https://arxiv.org/abs/2107.03374.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. Mind2web: Towards a generalist agent for the web, 2023. URL https://arxiv.org/abs/2306.06070.

Boyu Gou, Ruohan Wang, Boyuan Zheng, Yanan Xie, Cheng Chang, Yiheng Shu, Huan Sun, and Yu Su. Navigating the digital world as humans do: Universal visual grounding for gui agents, 2024. URL https://arxiv.org/abs/2410.05243.

Jiewen Hu, Thomas Zhu, and Sean Welleck. minictx: Neural theorem proving with (long-)contexts, 2024. URL https://arxiv.org/abs/2408.03350.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. ArXiv, abs/2406.00515, 2024. URL https://api.semanticscholar.org/CorpusID:270214176.

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, abs/2310.06770, 2023. URL https://api.semanticscholar.org/CorpusID:263829697.

Haolin Jin, Linghan Huang, Haipeng Cai, Jun Yan, Bo Li, and Huaming Chen. From llms to llm-based agents for software engineering: A survey of current, challenges and future. ArXiv, abs/2408.02479, 2024. URL https://api.semanticscholar.org/CorpusID:271709396.

Liqiang Jing, Zhehui Huang, Xiaoyang Wang, Wenlin Yao, Wenhao Yu, Kaixin Ma, Hongming Zhang, Xinya Du, and Dong Yu. Dsbench: How far are data science agents to becoming data science experts? arXiv preprint arXiv:2409.07703, 2024.

Jing Yu Koh, Robert Lo, Lawrence Jang, Vikram Duvvur, Ming Chong Lim, Po-Yu Huang, Graham Neubig, Shuyan Zhou, Ruslan Salakhutdinov, and Daniel Fried. Visualwebarena: Evaluating multimodal agents on realistic visual web tasks, 2024a. URL https://arxiv.org/abs/2401.13649.

Jing Yu Koh, Stephen McAleer, Daniel Fried, and Ruslan Salakhutdinov. Tree search for language model agents. ArXiv, abs/2407.01476, 2024b. URL https://api.semanticscholar.org/CorpusID:270870063.

Beck LaBash, August Rosedale, Alex Reents, Lucas Negritto, and Colin Wiel. Res-q: Evaluating code-editing large language model systems at the repository scale, 2024. URL https://arxiv.org/abs/2406.16801.

Grégoire Mialon, Clémentine Fourrier, Craig Swift, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants, 2023. URL https://arxiv.org/abs/2311.12983.

Niels Mündler, Mark Niklas Müller, Jingxuan He, and Martin Vechev. Swt-bench: Testing and validating real-world bug-fixes with code agents, 2025. URL https://arxiv.org/abs/2406.12952.

OpenAI. Introducing operator. OpenAI, 2025. https://openai.com/index/introducing-operator/.

Pranav Putta, Edmund Mills, Naman Garg, Sumeet Ramesh Motwani, Chelsea Finn, Divyansh Garg, and Rafael Rafailov. Agent q: Advanced reasoning and learning for autonomous ai agents. ArXiv, abs/2408.07199, 2024. URL https://api.semanticscholar.org/CorpusID:271865516.

Christopher Rawles, Alice Li, Daniel Rodriguez, Oriana Riva, and Timothy Lillicrap. Androidinthewild: A large-scale dataset for android device control. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (eds.), Advances in Neural Information Processing Systems, volume 36, pp. 59708–59728. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/bbbb6308b402fe909c39dd29950c32e0-Paper-Datasets_and_Benchmarks.pdf.

Chufan Shi, Cheng Yang, Yaxin Liu, Bo Shui, Junjie Wang, Mohan Jing, Linran Xu, Xinyu Zhu, Siheng Li, Yuxiang Zhang, Gongye Liu, Xiaomei Nie, Deng Cai, and Yujiu Yang. Chartmimic: Evaluating lmm's cross-modal reasoning capability via chart-to-code generation. ArXiv, abs/2406.09961, 2024. URL https://api.semanticscholar.org/CorpusID:270521907.

Chenglei Si, Yanzhe Zhang, Zhengyuan Yang, Ruibo Liu, and Diyi Yang. Design2code: How far are we from automating front-end engineering? ArXiv, abs/2403.03163, 2024. URL https://api.semanticscholar.org/CorpusID:268248801.

Jordan Sissel. xdotool: Fake keyboard/mouse input, window management, and more. https://github.com/jordansissel/xdotool. Accessed: 2025-02-12.

Mark Towers, Ariel Kwiatkowski, Jordan Terry, John U. Balis, Gianluca De Cola, Tristan Deleu, Manuel Goulão, Andreas Kallinteris, Markus Krimmel, Arjun KG, Rodrigo Perez-Vicente, Andrea Pierré, Sander Schulhoff, Jun Jet Tai, Hannah Tan, and Omar G. Younis. Gymnasium: A standard interface for reinforcement learning environments, 2024. URL https://arxiv.org/abs/2407.17032.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024a. URL https://arxiv.org/abs/2402.01030.

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, 2024b. URL https://arxiv.org/abs/2407.16741.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying llm-based software engineering agents, 2024. URL https://arxiv.org/abs/2407.01489.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024. URL https://arxiv.org/abs/2404.07972.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. Theagentcompany: Benchmarking llm agents on consequential real world tasks, 2024. URL https://arxiv.org/abs/2412.14161.

Jianwei Yang, Hao Zhang, Feng Li, Xueyan Zou, Chunyuan Li, and Jianfeng Gao. Set-of-mark prompting unleashes extraordinary visual grounding in gpt-4v, 2023a. URL https://arxiv.org/abs/2310.11441.

John Yang, Akshara Prabhakar, Karthik Narasimhan, and Shunyu Yao. Intercode: Standardizing and benchmarking interactive coding with execution feedback, 2023b. URL https://arxiv.org/abs/2306.14898.

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, 2024a. URL https://arxiv.org/abs/2405.15793.

John Yang, Carlos E. Jimenez, Alex L. Zhang, Kilian Lieret, Joyce Yang, Xindi Wu, Ori Press, Niklas Muennighoff, Gabriel Synnaeve, Karthik R. Narasimhan, Diyi Yang, Sida I. Wang, and Ofir Press. Swe-bench multimodal: Do ai systems generalize to visual software domains?, 2024b. URL https://arxiv.org/abs/2410.03859.

Shunyu Yao, Howard Chen, John Yang, and Karthik Narasimhan. Webshop: Towards scalable real-world web interaction with grounded language agents, 2023. URL https://arxiv.org/abs/2207.01206.

Daoguang Zan, Zhirong Huang, Ailun Yu, Shaoxin Lin, Yifan Shi, Wei Liu, Dong Chen, Zongshuai Qi, Hao Yu, Lei Yu, Dezhi Ran, Muhan Zeng, Bo Shen, Pan Bian, Guangtai Liang, Bei Guan, Pengjie Huang, Tao Xie, Yongji Wang, and Qianxiang Wang. Swe-bench-java: A github issue resolving benchmark for java, 2024. URL https://arxiv.org/abs/2408.14354.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. Autocoderover: Autonomous program improvement, 2024. URL https://arxiv.org/abs/2404.05427.

Boyuan Zheng, Boyu Gou, Jihyung Kil, Huan Sun, and Yu Su. Gpt-4v (ision) is a generalist web agent, if grounded. arXiv preprint arXiv:2401.01614, 2024.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents, 2024. URL https://arxiv.org/abs/2307.13854.

Table 4: Comparison of Hand-engineered Tools across Methods versus `PwP`. `PwP` natively supports all tools.

Method	**Hand-engineered Tools**	**Supported in `PwP`**
Agentless Xia et al. (2024)	File Edit, Repository Structure, File Structure	✓
CodeAct Wang et al. (2024a)	File Edit, IPython, Bash	✓
SWE-agent Yang et al. (2024a)	Search File, Search Text, File Edit	✓
EnIGMA Abramovich et al. (2024)	SWE-agent Tools + Debugger, Terminal, Connection Tool	✓
swebench-mm Yang et al. (2024b)	SWE-agent Tools + View Webpage, Screenshot, Open Image	✓

```python
bench = PwPBench(dataset='swebench')
# Replace with any dataset from PwP-Bench
dataset = bench.get_dataset()

# Set up environment and get initial observation
env = bench.get_env(dataset[0])
observation: PIL.Image = env.get_observation()['screenshot']

# Generate and execute action
action = agent.get_action(observation)
print(action)
# Output: xdotool mousemove 1000 1200
# click 1 && xdotool type 'hello world'
observation, info = env.step(action)

env.render()

# State Checkpointing
env.add_checkpoint('before_submit')
state = env.step('xdotool key Return')
env.resume_checkpoint('before_submit')

# Environment control
env.pause()
env.resume()

# Get reward and reset
is_success = env.get_reward()
env.reset()
```

Figure 5: Example demonstrating interaction with PwP environment, including keyboard/mouse actions, checkpointing, and state management. The code shows basic initialization, action execution, environment control, and reward handling.

A Programming with Pixels (PwP) Environment

A.1 Tools

Previous methods have proposed use of various hand-engineered tools. However, as shown in Table 4, `PwP` natively supports all these tools.

A.2 Example Interaction

Figure 5 shows an example of how to interact with `PwP` environment.

B Related Work

B.1 Task-specific SWE benchmarks

Early neural code generation approaches were typically evaluated on fixed input-output pairs—for example, generating code from docstrings Chen et al. (2021) or from general textual descriptions Austin et al. (2021). Subsequent benchmarks extended these evaluations to interactive settings, such as resolving GitHub pull requests or writing unit tests for real-world code repositories Jimenez et al. (2023); Zan et al. (2024); Mündler et al. (2025). More recently, efforts have broadened the scope of code generation to include multimodal tasks, where vision models must interpret images to generate correct code or edits Si et al. (2024); Shi et al. (2024); Jing et al. (2024); Yang et al. (2024b). However, each of these benchmarks is confined to specific languages, modalities, or task types. In contrast, our proposed `PwP-Bench` unifies these diverse evaluations into a single framework, encompassing multimodal and multilingual challenges that require extensive interaction with a broad suite of IDE tools. Using this unified approach we not only reproduce the performance of established benchmarks but also encourage the development of general-purpose agents capable of handling a superset of software engineering tasks. We show comparison of `PwP-Bench` with other datasets in Table 5.

B.2 SWE Agents

Recent works have explored "code agents" that move beyond single-step neural code generation toward interactive methods, where intermediate feedback from tools informs subsequent actions. However, many of these approaches specialize in particular tools or programming languages Jin et al. (2024); Yang et al. (2024b), limiting their broader applicability. For example, Agentless Xia et al. (2024) relies on a tool that parses files into Python-specific class and function structures, requiring additional adaptation and failing to perform well in other languages or settings Yang et al. (2024b). Similarly, SWE-agent requires modifications to adapt to different tasks Abramovich et al. (2024); Yang et al. (2024b). In contrast, agents designed for `PwP` are inherently task and language-agnostic due to the expressive action and observation spaces mandated by our environment. Moreover, the diverse tasks in `PwP-Bench` require agents to generalize across a wide range of SWE challenges rather than excel in one narrowly defined area.

Many existing agents also depend on hand-engineered tools that demand substantial human effort and are susceptible to bugs. For instance, Agentless Xia et al. (2024) leverages tools for parsing files into Python-specific structures; CodeAct relies on an IPython kernel Wang et al. (2024a); SWE-agent uses dedicated search and file editing tools Yang et al. (2024a); AutoCodeRover requires a linter Zhang et al. (2024); and SWE-agent EnIGMA develops specialized tools for CTF-style competitions Abramovich et al. (2024), while swebench-mm Yang et al. (2024b) employs a browser view. In `PwP`, most of these tools are inherently available within the IDE (as detailed in the Appendix A.1), and the agent's task is to effectively utilize them rather than being explicitly guided on which tool to use for each specific task.

Finally, current approaches often blur the line between the agent and the environment, as each agent is designed with its own specified action and observation spaces within a self-created environment. `Programming with Pixels` addresses this issue by unifying existing environments into a single, general-purpose platform on which agents operate. This clear separation of environment design from agent design standardizes evaluation and also allows any existing agent to be modeled within our framework, making it an important testbed for both current and future SWE agents.

B.3 Visual Agents and Computer-Use Agents

Several multimodal agent benchmarks have recently been proposed Koh et al. (2024a); Deng et al. (2023); Zheng et al. (2024) that require agents to operate user interfaces using a predefined, limited set of actions (e.g., `new_tab`, `go_back`, `click [element id]`). These *visual agents* typically rely on additional prompting—such as set-of-marks techniques that supply an HTML accessibility tree containing textual and positional information—to overcome their inherent poor visual grounding capabilities Yang et al. (2023a). Despite such aids, these agents often fail when faced with the complex and dense IDE interfaces found in our environment.

In contrast, *computer-use agents* Anthropic (2024); OpenAI (2025); Gou et al. (2024) are trained to operate with an expressive action and observation space using primitive operations like clicks and keystrokes, without the need for external accessibility elements. However, there has been a lack of a SWE-specific environment for evaluating and further training these agents. `PwP` fills this gap by providing a unified, expressive IDE platform that challenges computer-use agents with realistic and diverse SWE tasks.

B.4 Expressive Agent Environments

Prior work on expressive agent environments has predominantly targeted the web domain Koh et al. (2024a); Deng et al. (2023), entire operating systems Xie et al. (2024); Bonatti et al. (2024); Rawles et al. (2023), or other general scenarios Xu et al. (2024). Some of these environments, such as OSWorld, feature general action and observation spaces similar to ours. However, although these benchmarks are capable of expressing a wide range of tasks, they do not focus on the unique challenges inherent to software engineering within an IDE. For example, while OSWorld Xie et al. (2024) offers a broad set of tasks, it is not specifically designed for SWE, resulting in increased computational overhead. Software engineering is a diverse and important domain that merits its own dedicated environment.

In contrast, we design `PwP` so that existing agents can be readily incorporated into our framework. Specifically, we modify the sourcecode of IDE to facilitate developement of tools previously used by agents directly through an API call instead of user interaction. These specific modifications to the IDE enable agents to interact effectively with the environment and gain direct access to IDE state information—facilitating both tool utilization and robust evaluation. Furthermore, `PwP-Bench` is tailored specifically for multimodal SWE tasks within an IDE, encompassing activities such as pull-request handling, debugging, and image-based code generation across multiple programming languages. We also observe that existing agents built for generic UI control often struggle in the `PwP` environment, as they must interact with a richer set of tools and achieve precise visual grounding within a complex interface containing a large number of interactive elements. We compare `PwP` with other environments in Table 1.

C PwP-Bench

Metrics We use individual metrics mentioned in original datasets. When reporting results on `PwP-Bench`, we report marco average of all these metrics. In particular 11/15 using Accuracy as their metric. However, due to complexity of dataset, these often goes beyond simple accuracy metric and in some cases, the dataset is evaluated on multiple orthogonal metrics, instead of one. We detail, these metrics for each of the datasets.

- **SWT-Bench** evaluates generated tests by the agent, and reports 6 different metrics: Applicability, Success Rate, F- X, F- P, P- P and Coverage. We report average of all 6 metrics.
- **ChartMimic** evaluates generated code on various metrics such as accuracy of text, colors used, legend etc. We average all metrics similar to original dataset.
- **Design2Code** evaluates generated code on various metrics such as accuracy of text, position, clip score, etc. We average all metrics similar to original dataset.
- **DSBench** has two categories, one containing MCQ questions, while other containing generating code for Kaggle Competitions. We use 10/10 instances from each category in `PwP-Bench`-Lite. While MCQ questions are evaluated using Accuracy, the code generation part is evaluated using linear normalization between baseline score (of the competition) and the score of the winner of competition.

D Results

Table 6 shows results for all agent designs on each of 15 datasets in `PwP-Bench`. Along with the results of our two categories of agents, we also include previously state-of-the-art results for comparison. We make our best effort to include latest publicly available results, however, there may be minor discrepancies.

Table 5: **Comparison of existing software engineering benchmarks**. `PwP-Bench` provides the largest dataset (5400 instances) and uniquely covers all aspects: multiple languages and modalities, real IDE interaction, interactive coding, and both code generation and general software engineering tasks.

Benchmark	#Instances	Multiple Languages	Multiple Modalities	Real IDE Env	Interactive Coding	Non-Code SWE Tasks	Code-Generation SWE Tasks
SWE-Bench Jimenez et al. (2023)	2K	✗	✗	✗	✓	✗	✓
SWE-Bench-MM Yang et al. (2024b)	$\leq 1K$	✗	✓	✗	✓	✗	✓
LiveCodeBench Jain et al. (2024)	$\leq 1K$	✗	✗	✗	✓	✗	✓
Aider Polyglot Aider (2024)	$\leq 1K$	✓	✗	✗	✓	✗	✓
TheAgentCompany Xu et al. (2024)	$\leq 1K$	✗	✓	✓	✓	✓	✗
VisualWebArena Koh et al. (2024a)	$\leq 1K$	✗	✓	✗	✗	✗	✗
OSWORLD Xie et al. (2024)	$\leq 1K$	✗	✓	✓	✗	✓	✗
WindowsAgentArena Bonatti et al. (2024)	$\leq 1K$	✗	✓	✓	✗	✓	✗
`PwP-Bench` (Ours)	5400	✓	✓	✓	✓	✓	✓

Table 6: Performance Evaluation of Different Models Across Task Categories. Leged: HE: HumanEval, SB: SWEBench, SJ: Swebench-Java, RQ: ResQ, CI: CaniteEdit, ST: SWTBench, DC: Design2Code, CM: ChartMimic, DS: DSBench, SM: Swebench-MM, IC: Intercode-CTF, BD: Bird SQL, MC: Minictx, VS: VSCode, GS: No-Code SWE Tasks. *Much more costly method

	Code Generation & Editing						**Multimodal Code Generation**				**Domain-Specific Code Generation**			**No-Code SWE Tasks**		**Overall**
Model	HE	SB	SJ	RQ	CI	ST	DC	CM	DS	SM	IC	BD	MC	VS	GS	**Avg**
Computer-Use Agents (Screenshot + SoM)																
GPT-4o	5%	0.0%	0.0%	0.0%	0.0%	0.0%	48.7%	0.7%	16.1%	0.0%	5.0%	0.0%	0.0%	20.0%	0.0%	6.4%
GPT-4o-mini	0.0%	0.0%	0.0%	0.0%	5.0%	0.0%	14.8%	0.0%	17.2%	0.0%	0.0%	0.0%	0.0%	5.0%	0.0%	2.8%
Gemini-Flash	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	15.2%	2.0%	25.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	2.8%
Gemini-Pro	10.0%	0.0%	0.0%	0.0%	5.0%	0.0%	14.5%	8.1%	25.0%	0.0%	0.0%	0.0%	0.0%	15.0%	0.0%	5.2%
Claude-Sonnet	20.0%	0.0%	0.0%	15.0%	25.0%	4.2%	18.1%	0.0%	50.0%	10.0%	15.0%	0.0%	0.0%	35.0%	5.0%	13.2%
Computer-Use Agents (Screenshot + SoM + File/Bash Operations)																
GPT-4o	85%	25%	15%	30%	50%	17.0%	70.2%	65.5%	36.6%	**20%**	70%	10%	5%	10%	0.0%	34.0%
GPT-4o-mini	60%	10%	5%	20%	30%	16.7%	41.3%	5.5%	39.6%	15%	40%	5%	0%	10.0%	0.0%	19.9%
Gemini-Flash	0	5%	5%	15%	15%	17.1%	19.9%	13.5%	36.0%	10%	25%	0%	0%	5%	0.0%	11.1%
Gemini-Pro	85%	10%	15%	15%	40.0%	20.2%	25.6%	24.7%	33.6%	15%	5%	5%	0%	10%	0.0%	20.3%
Claude-Sonnet	**95%**	25%	**35%**	**55%**	**65%**	**37.4%**	83.4%	71.2%	66.3%	10%	**100%**	15%	15%	**35%**	5.0%	**47.2%**
Previous State of the Art Reported*																
	98.8%	55%*	9.9%	58%	63.3%	$\approx 37\%$	90.2%	71.4%	44.6%	12.2%	72%	30.2%	-	-	-	42.8%*

D.1 Analysis

Agents Fail to Edit Files. The deficiencies in visual grounding significantly impact the file editing capabilities of VLMs. Even when provided with cursor location information in textual form, these models struggle to interpret such data amid complex UI elements. Models fine-tuned for UI interactions still commit basic editing errors—such as incorrect indentation and text misplacement—and are unable to recover from these errors (see Appendix for examples). We speculate these limitations could stem from two factors: (i) model overfitting to user interafaces in their training domains, or (ii) the increased complexity of the `PwP` IDE interface, which contains substantially more interactable elements than typical web or OS environments. Addressing these limitations represents an important direction for future work. Although direct file access via tool operations is available, UI-based editing confers unique advantages for tasks such as editing Jupyter notebooks, comparing changes, or modifying specific sections of large files. These results underscore two limitations: (i) current VLMs are challenged by complex UI interactions beyond simple web/OS interfaces Xie et al. (2024); Koh et al. (2024a), and (ii) the inability to effectively perform UI-based editing prevents agents from leveraging valuable IDE features that could have improved their performance.

Agents Are Incapable of Recovering from Errors. Our analysis indicates that agents—especially those based on smaller models—demonstrate limited error recovery capabilities. When an action fails to execute correctly, models tend to persistently repeat the same failed action without exploring alternatives. Similarly, if an agent selects an incorrect action, it continues along an erroneous solution path without recognizing or correcting the mistake. In an experiment designed to further probe this behavior, we deliberately suppressed one of the model's actions. Despite the environment's screenshot clearly showing an unchanged state, the models proceeded with their planned action sequence as though the suppressed action had succeeded. This behavior suggests a heavy reliance

Figure 6: **Example of Agent Missing Visual Error Indicators** The agent fails to recognize linter error indicators (wavy underlines).

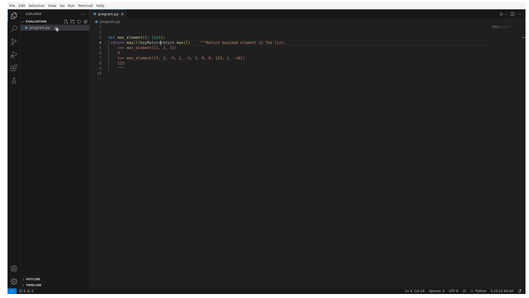

Figure 7: **Example of Agent's Inability to Perform File Editing** The agent incorrectly positions new content in the file editor.

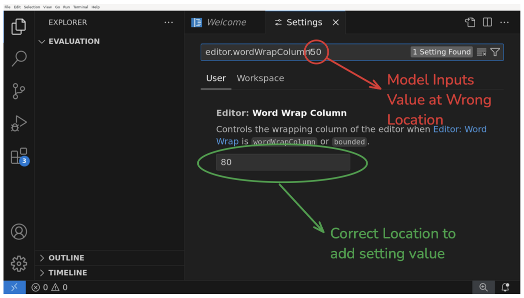

Figure 8: **Example of Agent Misidentifying UI Elements** The agent fails to identify the correct input field, typing '50' into the settings search bar instead of the word wrap column setting field.

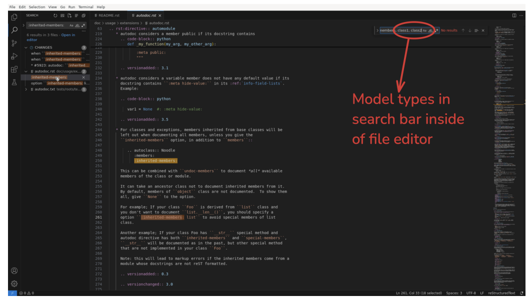

Figure 9: **Example of Agent Misidentifying Active Panel** The agent fails to recognize the active editor panel, incorrectly typing into the search bar (red arrow) instead of the file editor.

on memorized action sequences rather than dynamic responses to visual feedback, resulting in exponentially increasing error propagation and ultimately poor performance. This inablity of current agent's not being able to recover from errors, results in exponentially increasing error propogation, resulting in poor performance.

E Discussion

Why use IDE over simple Bash Agent? A natural question, that arises is given that `PwP` requires more complex visual grounding capabilities, why not use simple bash scripts for all task. The shorter answer, is because modern IDEs, have been developed over multiple years of effort, and provide several advantages that are not possible with bash interface. While, theoretically it may still be possible to create equivalent tools, it would take similar tremendous effort, to develop them again for agents, with less reliability.

To give few examples of myriads advantages of IDEs:

- **Interactive Debugging Capabilities**
  - IDEs provide rich, stateful debugging interfaces that allow AI agents to set breakpoints, inspect variables, and evaluate expressions dynamically
  - Unlike CLI debuggers (GDB, LLDB, pdb), IDE debuggers maintain visual context and state, making it easier for AI agents to track program flow and debug complex scenarios
  - The visual representation of stack traces and variable states is more structured and machine-parseable compared to text-based CLI output
- **Intelligent Code Refactoring**

- IDEs maintain a complete Abstract Syntax Tree (AST) of the project, enabling accurate symbol renaming and code restructuring across multiple files
- AI agents can leverage IDE's semantic understanding to perform complex refactoring operations with higher confidence
- Unlike text-based search-and-replace in Bash, IDE refactoring tools understand code context and prevent accidental modifications to unrelated symbols

- **Test Management and Coverage Analysis**
  - IDEs provide structured APIs for test discovery, execution, and result analysis
  - AI agents can efficiently track test coverage through visual indicators and programmatic interfaces
  - Real-time test feedback and coverage data is more readily accessible compared to parsing CLI test runner output
- **Performance Profiling and Analysis**
  - IDE profilers offer structured data about CPU usage, memory allocation, and runtime behavior
  - Visual representations of performance metrics (flame graphs, memory usage) are easier for AI agents to analyze systematically
  - Profiling data is available through APIs rather than requiring parsing of complex text-based output
- **Code Indexing and Semantic Search**
  - IDEs maintain comprehensive code indexes that enable fast, context-aware code search and navigation
  - AI agents can leverage these indexes for more accurate code understanding and modification
  - Unlike grep or find, IDE search capabilities understand code structure and can filter based on semantic properties
- **Extension Integration and Automation**
  - IDE extensions can be programmatically controlled through APIs, allowing AI agents to leverage additional tools seamlessly
  - Extensions can provide structured data and interfaces that are more reliable for automation compared to parsing CLI tool output
  - Configuration and coordination of multiple tools can be managed through unified IDE interfaces rather than managing separate CLI tools