ToM-SWE: User Mental Modeling for Software Engineering Agents

Anonymous authors
Paper under double-blind review

Abstract

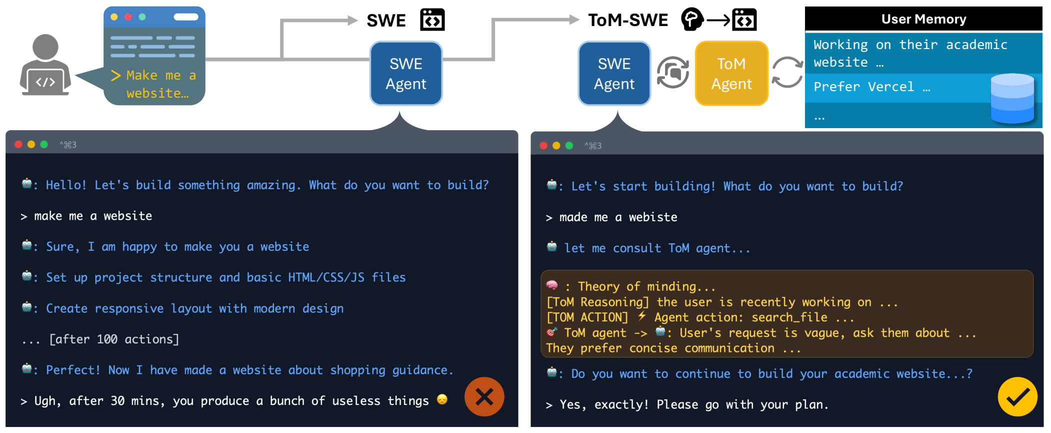
Recent advances in coding agents have made them capable of planning, editing, running, and testing complex codebases. Despite their growing ability in coding tasks, these systems still struggle to infer and track user intent, especially when instructions are underspecified or context-dependent. To bridge this gap, we introduce ToM-SWE, a dual-agent architecture that pairs a primary software-engineering (SWE) agent with a lightweight theory-of-mind (ToM) partner agent dedicated to modeling the user's mental state. The ToM agent infers user goals, constraints, and preferences from instructions and interaction history, maintains a **persistent memory** of the user, and provide user-related suggestions to the SWE agent. In two software engineering benchmarks (ambiguous SWE-bench and stateful SWE-bench), ToM-SWE improves task success rates and user satisfaction. Notably, on stateful SWE benchmark, a newly introduced evaluation that provides agents with a user simulator along with previous interaction histories, ToM-SWE achieves a substantially higher task success rate of 59.7% compared to 18.1% for OpenHands, a state-of-the-art SWE agent. Furthermore, in a three-week study with professional developers using ToM-SWE in their daily work, participants found it useful 86% of the time, underscoring the value of stateful user modeling for practical coding agents.

1 Introduction

Recent advances in large language models (LLMs) have enabled coding agents to perform complex software engineering tasks, from code generation (Jiang et al., 2024) and debugging (Tian et al., 2024) to system design (Kovacic et al., 2025) and optimization (Gao et al., 2024). However, despite their impressive technical capabilities, coding agents often struggle with a fundamental aspect of software development: *effective communication* and *collaboration* with human developers.

The core limitation is that current systems lack explicit mechanisms for modeling and predicting human intent in long-horizon, multi-turn interactions (Kim et al., 2023). Unlike human developers who naturally build mental models of their collaborators' goals, preferences, and constraints through various tasks (Tomasello, 2009), coding agents lack the mechanism to infer and acquire the underlying user intentions from the surface-level, which in real-world interactions are often ambiguous, incomplete, or context-dependent (Levinson, 1983). Furthermore, current coding agents typically operate in a stateless manner, treating each session as independent rather than maintaining persistent context about the user's evolving goals and conversation history. This paradigm often leads to wasted effort and misunderstandings, and in high-stakes settings can result in erroneous, or even unsafe outcomes.

To bridge the gap between current coding agents and the challenges of inferring user intent in long-horizon interactions, we introduce **ToM-SWE**, a conceptual framework that integrates *theory-of-mind* (ToM) reasoning into software engineering agents. Here, ToM refers specifically to the ability to model a user's mental state, including goals, preferences, and intentions, based on user instructions and interaction history. As shown in Figure 1, ToM-SWE operationalizes this idea through a dual-agent architecture: a primary software engineering (SWE) agent remains focused on coding tasks, while a dedicated ToM agent models the user's mental state and supports the SWE agent when needed. This separation is crucial for two reasons: it preserves the SWE agent's coding per-



```
: Hello! Let's build something amazing. What do you want to build?
> make me a website
: Sure, I am happy to make you a website
: Set up project structure and basic HTML/CSS/JS files
: Create responsive layout with modern design
... [after 100 actions]
: Perfect! Now I have made a website about shopping guidance.
> Ugh, after 30 mins, you produce a bunch of useless things
```

```
: Let's start building! What do you want to build?
> made me a webiste
 let me consult ToM agent...
 : Theory of minding...
[ToM Reasoning] the user is recently working on ...
[TOM ACTION] ⚡ Agent action: search_file ...
 ToM agent -> : User's request is vague, ask them about ...
They prefer concise communication ...
: Do you want to continue to build your academic website...?
> Yes, exactly! Please go with your plan.
```

Figure 1: An example of how the ToM-SWE framework can help the SWE agent to model the user's mental state and provide more useful suggestions based on the previous interaction history. When facing the same starting instruction ("Make me a website..."), the SWE agent simply generates code yet fails to meet the users' requirement for the task. However, with the ToM agent, the SWE agent can first consult the ToM agent that persists user mental state across multiple sessions, and then act more aligned with the user's preferences and constraints.

formance, and it enables specialized, persistent user modeling that developers can flexibly invoke and customize for efficiency and privacy. The ToM agent itself functions in two complementary modes to keep track of the user's preferences, emotions, and etc. During *active coding sessions* (in-session ToM), it infers the user's underlying mental state (e.g., the "true" intent behind potentially ambiguous instructions). After each session, it works to create mental models of the user (after-session ToM), consolidating interaction history to refine its beliefs about the user's mental state in a hierarchical way.

To evaluate ToM-SWE's effectiveness, we introduce the **Stateful SWE benchmark**, the first benchmark that allows agents to leverage realistic conversation histories across multiple coding sessions to track user mental state. In this setting, agents interact with an LLM-powered user simulator and receive synthesized interaction histories to guide their reasoning. To stay consistent with prior evaluation, we borrow the original issues from SWE-bench (Chowdhury et al., 2024), reframing them as casual starting instructions paired with different user profiles and interaction histories. The agent needs to understand user intent, preferences and constraints either through interacting with the LLM-powered user simulator or inferring from the past interaction histories. Unlike existing benchmarks such as SWE-bench, which primarily assess technical problem-solving, Stateful SWE evaluates an agent's ability to sustain meaningful interactions over time. While Ambiguous SWE-bench (Vijayvargiya et al., 2025) tests ambiguity resolution by underspecifying instructions, it does not capture long-term memory demands.

We evaluate our ToM-SWE framework on both our newly introduced stateful SWE benchmark as well as the stateless Ambiguous SWE-bench (Vijayvargiya et al., 2025), building our agent (`ToMCodeAct`) using the OpenHands platform (Wang et al., 2025), an open-source framework for developing SWE agents. We show that **`ToMCodeAct` outperforms the OpenHands SOTA `CodeAct` agent** (Wang et al., 2024) on both benchmarks. On the ambiguous SWE-bench, `ToMCodeAct` agent achieves 63.4% issue resolved rate compared to CodeAct agent's 51.9% (**+11.5% improvement**). On the stateful SWE-bench, `ToMCodeAct` agent achieves 57.4% (**+43.9%**) task resolved rate compared to CodeAct agent's 13.5%. Furthermore, `ToMCodeAct` agent achieves substantially better user satisfaction scores of 3.62 compared to CodeAct agent's 2.57 (**+41% improvement**, automatically measured through user simulators that evaluate preference alignment, communication, etc.). The results highlight the importance of modeling user in real-world software development scenarios to respect specific user preferences and constraints beyond task completion.

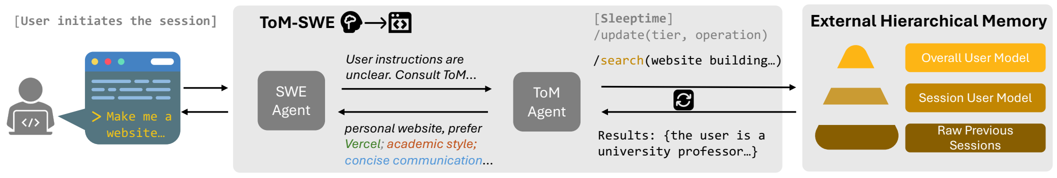

[User initiates the session]
> Make me a website…
ToM-SWE
SWE Agent
User instructions are unclear. Consult ToM...
personal website, prefer Vercel; academic style; concise communication...
ToM Agent
[Sleeptime]
/update(tier, operation)
/search(website building…)
Results: {the user is a university professor…}
External Hierarchical Memory
Overall User Model
Session User Model
Raw Previous Sessions

Figure 2: Overview of the ToM-SWE framework: the SWE agent handles code generation and execution, while the ToM agent focuses on user modeling and intent inference. The SWE agent consults the ToM agent to predict the user's mental state before suggesting technical actions. Meanwhile, the ToM agent maintains an external hierarchical memory system to persist the user's state and update user models after each session (with `update_memory` action).

Finally, we conduct an *in-the-wild* human study with professional developers to validate the practical effectiveness of our approach and **find that ToM agent's suggestions are useful 86% of the time.** Unlike previous work that conducts human studies on predefined tasks (Wu et al., 2025; Qian et al., 2025), we recruit 17 software developers to use the ToM-enhanced OpenHands CLI (Wang et al., 2025) for their everyday, self-chosen coding tasks over three weeks. Each time the SWE agent communicates with the ToM agent, the developers choose to accept, partially accept, or reject the ToM agent's suggestions. Developers can also provide feedback through a public Slack channel anytime during the study. Besides the high acceptance rate, we learn from the developers' feedback that ToM agent can often provide useful and valuable suggestions that make users workflow more efficient (e.g., *"Please add pytest for the new function"* or *"keep your code edit minimal"*). Furthermore, ToM agent can even provide novel and preference aligned suggestions (e.g., *"refactor the code following Linux development philosophy"*).

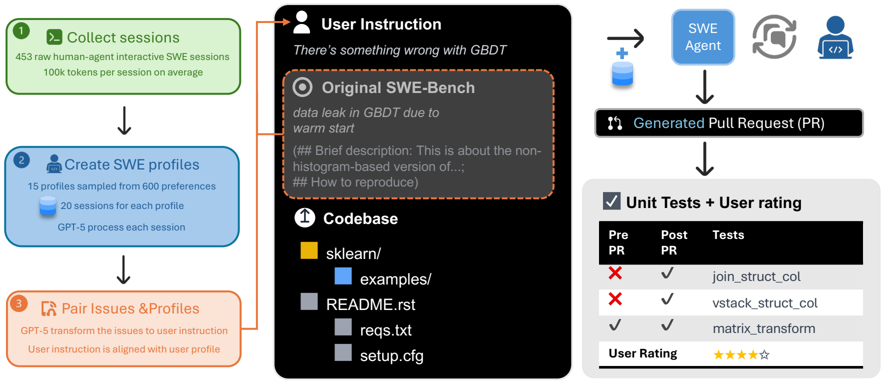
2 ToM-SWE: Pairing SWE Agent with ToM Agent

Consider existing setups of software engineering agents such as SWE-agent (Yang et al., 2024) and OpenHands CodeAct (Wang et al., 2024; 2025). At time step t in coding session i, an agent receives an observation $o_t \in O$ either from the environment (e.g., terminal output, file contents, test results) or the user (e.g., user instructions, feedback). Then the agent takes an action $a_t \in A$ (e.g., code edits, shell commands) following some policy $\pi(a_t|c_t^i)$, where $c_t^i = (o_1, a_1, \ldots, o_{t-1}, a_{t-1}, o_t)$ is the context available to the agent in the coding session i until time step t.

The mapping $c_t^i \mapsto a_t$ becomes challenging when accurate execution of the tasks requires understanding implicit user preferences as such information often exist in past sessions $\{c^j\}_{j=1}^{i-1}$, instead of the current context c_t^i. For example, when a user says *"implement a web scraper"*, the agent needs to infer library preferences (`requests` vs `httpx`), which could only be available from previous interactions. To bridge this gap, we introduce a theory-of-mind (ToM) agent that explicitly models the user's mental state (Figure 2). In the following, we explain (1) how the SWE agent queries the ToM agents for relevant information and (2) how the ToM agent models the user's mental state.

SWE Agent Interaction with ToM Agent We enable the SWE agent to interact with the ToM agent by introducing two additional tools into its available toolset: (1) `consult_tom` (**in-session**): the SWE agent sends a query q and the current session context c_t^i to the ToM agent (action a_t), the ToM agent outputs relevant user mental state information m_{user} by reasoning over the interaction history $\{c^j\}_{j=1}^{i-1} \cup \{c_t^i\}$. This user modeling information is then incorporated into the SWE agent's context as $c_t \,\|\, [a_t, m_{user}]$, helping the agent to make decisions that aligns with the user's implicit preferences and constraints. (2) `update_memory` (**after-session**): After the SWE agent finishes the coding session, it uses this tool to inform the ToM agent to process the current session and update the hierarchical memory system.

ToM Agent Design The ToM agent models user mental states through a three-tier hierarchical memory system implemented as external database: (tier 1) raw session storage, stores complete previous session histories. (tier 2) session-based user model, maintains per-session analysis including


1
Collect sessions
453 raw human-agent interactive SWE sessions
100k tokens per session on average
2
Create SWE profiles
15 profiles sampled from 600 preferences
20 sessions for each profile
GPT-5 process each session
3
Pair Issues &Profiles
GPT-5 transform the issues to user instruction
User instruction is aligned with user profile
User Instruction
There's something wrong with GBDT
Original SWE-Bench
data leak in GBDT due to warm start
(## Brief description: This is about the non-histogram-based version of...;
## How to reproduce)
Codebase
sklearn/
examples/
README.rst
reqs.txt
setup.cfg
SWE Agent
Generated Pull Request (PR)
Unit Tests + User rating

Pre PR	Post PR	Tests
✗	✓	join_struct_col
✗	✓	vstack_struct_col
✓	✓	matrix_transform
User Rating		★★★★☆

Figure 3: Overview of the stateful SWE benchmark. We first **collect profiles** following the three steps outlined above. We then **create instances** by pairing the user simulator of different profiles with SWE-bench issues. Original SWE-bench issue descriptions are converted into vague user instructions (similar to Ambiguous SWE-bench (Vijayvargiya et al., 2025)), but our user simulator conditions on personalized profiles with distinct interaction styles and coding preferences, whereas Ambiguous SWE-bench uses fixed user simulator prompts. The agent solves the task under the same environment and tests of the original instance with access to the previous interaction histories with the user and could interact with the simulated user for extra information.

session intent, interaction patterns, and coding preferences. (tier 3) overall user model, aggregates cross-session patterns into preference clusters, interaction style summary, and coding style summary.

During the **in-session** phase, once the SWE agent sends the query and current session history (c_t^i) to the ToM agent, the ToM agent with overall the user model loaded in the context window could decide to use the `search_file` action to retrieve the relevant context or use the `read_file` action for a specific file from the (tier 1) and (tier 2) of the memory system. The retrieved/read content will be added to the context window of the ToM agent. The ToM agent can perform multiple actions to obtain the relevant information before providing suggestions to the SWE agent with `give_suggestions` action.

During the **after-session** phase, the ToM agent processes new session data through a structured workflow. Raw sessions are first added to (tier 1) automatically. The ToM agent then uses `analyze_session` with raw session data as the input, extracting user intent, emotional states, and message-level preferences to create structured session-based user models (tier 2). If there's no overall user model, ToM agent will use `initialize_user_model` to aggregate these session-based user models to update the overall user model (tier 3). If there's already an overall user model, ToM agent will use `update` action to update the overall user model. (See Appendix A.3 for more action space details.)

This dual-agent design offers two advantages over having the SWE agent handle user modeling directly: (1) **reduced context distraction**: the SWE agent maintains focus on technical tasks without being overwhelmed by extensive user history, (2) **specialized optimization**: each agent can be optimized for its specific domain (coding vs. user modeling)

3 STATEFUL SWE BENCHMARK

Most of previous SWE benchmarks solely focus on task completion (Zhao et al., 2024; Zan et al., 2025; Chowdhury et al., 2024). Here, we introduce a new benchmark that extends SWE-bench (Jimenez et al., 2024) to test agents' ability to interact, model and adapt to users while solving tasks. For each instance in our benchmark, agents not only have access to the coding environment, but can also interact with the simulated user, and check the previous session history with the user.

Profile Collection: As shown in Figure 3, we begin by collecting 453 real, consented sessions between human software developers and coding agents through the OpenHands platform (Wang et al., 2025). From the collected sessions, we derive 15 developer profiles capturing distinct **interaction**

styles (verbosity, question timing, response style) and **coding preferences** (testing practices, documentation habits, architectural choices). Each profile represents a unique combination of interaction patterns paired with coding preference clusters derived from 75 recurring practices observed in the sessions (see Appendix A.5 for detailed breakdown). For each profile, we randomly sample 20 sessions from the same developer to form the profile's history. To maintain realism, these sessions are processed with GPT-5 to align user messages with their corresponding profile characteristics (e.g., rephrasing a verbose user message into a concise one if the profile prefers concise exchanges).

Instance Creation and Evaluation: We pair created 15 developer profiles with 500 instances from the verified SWE-bench issues (Chowdhury et al., 2024) and run a user simulator powered by LLM that enables realistic human-agent interaction evaluation, inspired by Ambiguous SWE-bench Vijayvargiya et al. (2025). Differing from Ambiguous SWE-bench, the user simulator in Stateful SWE-bench is conditioning on the unique user profile with interactional and coding preferences. Therefore, simulators with different profiles behave differently, posing challenges for the agent to interact with different users. For example, a user with low verbosity preference will only answer one question and could express dissatisfaction or even refuse to answer if the agent asks too many questions in a single turn. Meanwhile, agents have access to previous conversation histories with the same user profile, requiring them to derive user preferences from past interactions to communicate successfully. Besides the standard SWE-bench instance evaluation, we could also apply the user simulator to evaluate the agent's ability to interact with the user. We give the full session data to the corresponding profile-conditioned user simulator and ask the user simulator to rate the agent from 1 to 5 and obtain the **user simulator satisfaction** scores (See Appendix A.7.2 for details).

4 Offline Benchmark Experiments

4.1 Experiment Setup

We implement all experiments using the OpenHands platform (Wang et al., 2025), an open-source framework for developing and evaluating AI software development agents. OpenHands provides sandboxed environments for safe code execution and standardized interfaces for agent-environment interaction while maintaining isolation and reproducibility. Our experiments build upon the `CodeAct` agent architecture (Wang et al., 2024), which uses executable code as a unified action space for agent interactions `CodeAct` consolidates traditional agent actions (e.g., file editing, command execution, web browsing) into executable code snippets that are dynamically interpreted within the environment.

Baselines and Setup Our experiments involve three agent variants: (1) `CodeAct` agent: The baseline implementation following Wang et al. (2024), which operates without explicit user modeling capabilities. (2) `RAGCodeAct` agent: An enhanced version that encourages the coding agent to proactively retrieve relevant information from previous interaction history, serving as the single agent paradigm to manage both the coding task and user modeling task simultaneously. (3) `TomCodeAct` agent: Our proposed approach that pair coding agent with a theory-of-mind agent, maintaining explicit user mental models and adapting behavior accordingly. We evaluate all agents using three state-of-the-art language models: Claude Sonnet 4, Claude 3.7 Sonnet and Qwen3-480B (Qwen3). All agents interact with the same simulated users and are prompted to ask for clarifications when uncertain. We use Claude Sonnet 4 for the theory-of-mind agent across all experiments and use BM25 to retrive relevant information for both the `RAGCodeAct` agent and the `TomCodeAct` agent (see Appendix A.7 for complete model specifications and hyperparameters).

Evaluation Benchmarks As described in Section 3, we evaluate our approach on two complementary benchmarks: the Ambiguous SWE benchmark and our Stateful SWE benchmark. Both benchmarks use GPT-5 powered simulated users and evaluate agents over 500 instances with a maximum of 100 interaction turns per task. Note that the user simulator has access to both the complete issue description and the hints for the issue from the original SWE-bench issues. Together, these benchmarks evaluate complementary scenarios: ambiguous formal specifications requiring clarification versus informal user instructions with available context. We report task **resolved rates** for both benchmarks and additionally report **user simulator satisfaction** scores for the Stateful SWE benchmark (Table 1).

4.2 Benchmark Results

We present evaluation results demonstrating ToM-SWE's effectiveness for both benchmarks and human study. Our findings show consistent improvements in both task resolution rates and user satisfaction when agents incorporate theory of mind capabilities for user modeling.

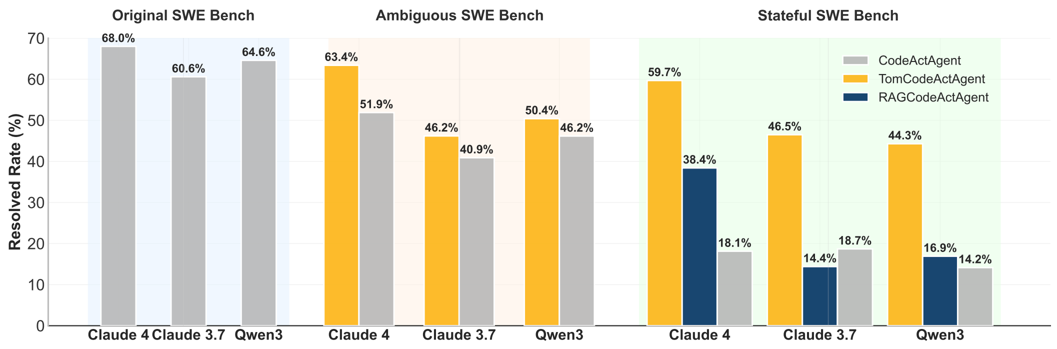


Figure 4: Performance comparison based on task resolved rates (measured by passed unit tests provided in the SWE-bench issues). `TomCodeAct` agent consistently outperforms `CodeAct` agent across both benchmarks and model variants, with the largest performance gap observed in the Stateful SWE benchmark using Claude Sonnet 4.

Figure 4 shows the resolved rates across all model-agent combinations. For example, `TomCodeAct` agent maintains its lead with 63.4% resolution rate using Claude Sonnet 4 versus `CodeAct` agent's 51.9% on the Ambiguous SWE benchmark. And `TomCodeAct` agent achieves 59.7% resolution rate with Claude Sonnet 4 compared to `CodeAct` agent's 18.1%, representing a 41.6 percentage point improvement on the Stateful SWE benchmark. Table 1 shows user satisfaction scores for the Stateful SWE benchmark. `TomCodeAct` agent achieves the highest satisfaction ratings across all models.

Table 1: User Simulator Satisfaction Scores on Stateful SWE Benchmark

Agent	Claude 3.7	Claude 4	Qwen3
`CodeAct`	$2.26_{\pm 0.08}$	$2.57_{\pm 0.08}$	$2.48_{\pm 0.08}$
`+RAG`	$2.32_{\pm 0.08}$	$3.09_{\pm 0.09}$	$2.54_{\pm 0.11}$
`+ToM`	$\mathbf{3.29}_{\pm 0.08}$	$\mathbf{3.62}_{\pm 0.07}$	$\mathbf{3.24}_{\pm 0.09}$

Besides the success of the `TomCodeAct` agent, we additionally have the following findings: (1) **Theory-of-mind reasoning over past user-agent interactions is essential** for understanding current user intent while simply retrieving raw session data provides limited help. We observe that while `RAGCodeAct` agent also has access to the previous raw session data, it still underperforms the `ToMCodeAct` agent across all models. This is especially true when the base model for SWE agent is not powerful enough to handle both the coding task and the user modeling task simultaneously. With Claude 3.7 Sonnet, `RAGCodeAct` agent even hurts the performance of the SWE agent (task resolved rate drops from 18.7% to 14.4%).

(2) **User satisfaction does not equal solving the task**. From Table 1, we observe the mismatch between task resolution rate and user satisfaction between `RAGCodeAct` agent and `CodeAct` agent. To further investigate the relationship between task resolution rate and user satisfaction, we separate the user satisfaction scores into three categories: **High** (3.5-5), **Medium** (2-3.5) and **Low** (1-2). And we focus on the cases where resolving the task disagrees with the user satisfaction, i.e., failed to resolve the task but user satisfaction is high (F+H) and resolved the task but user satisfaction is medium and low (S+M and S+L). Figure 5 provides a detailed analysis of agent performance across different categories along with the example feedback from the user simulator in Table 2. We observe that the `ToMCodeAct` agent has the highest F+H rates, indicating that even if the task is not resolved, the `ToMCodeAct` agent can still achieve higher user satisfaction. It is also interesting to see that the `RAGCodeAct` agent has the highest S+M rates, suggesting that modeling user preferences done wrong can lead to worse user satisfaction even if the task is resolved.

Table 2: Examples of Task Resolution and User Satisfaction Mismatches

Category	**Reason**	**User Simulator Example Feedback (`SWE-bench issue ID`)**
Fail + High	Meaningful progress	"precise fix with minimal back-and-forth, aligned well with my preferences" (`matplotlib__matplotlib-22865`)
	Good communication	"Asked for all key details up front (version, minimal repro, affected APIs), matching the preferred workflow" (`django__django-16256`)
Success + Med	Ignored user's preferred tools	"Ignored the preferred typing style... Did not provide a descriptive commit message" (`sympy__sympy-22456`)
	Poor communication style	"You didn't provide a brief summary..." (`scikit-learn__scikit-learn-13779`)

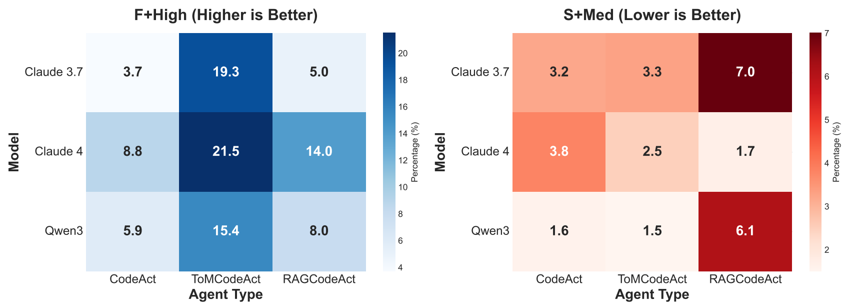


Figure 5: Task resolution and user satisfaction disagreement. F+H: the proportion of cases where the user simulator statisfaction score is high given the task is not resolved. S+M: the proportion of cases where the user simulator statisfaction score is medium or low given the task is resolved. **Takeaway:** ToMCodeAct agent can achieve higher user satisfaction even if the task is not resolved.

Understanding the Satisfaction-Resolution Relationship While user satisfaction and task resolution are distinct metrics, they exhibit a strong positive correlation. For the Claude 4 `ToMCodeAct` agent, Pearson correlation analysis reveals $r = 0.74$ ($p < 0.001$), indicating a strong relationship. Task resolution explains 55% of the variance in satisfaction scores ($R^2 = 0.55$), demonstrating that successful task completion remains the dominant driver of user satisfaction.

We then manually examine the 44 cases where the Claude 4 `ToMCodeAct` agent received high user satisfaction despite failing the task, our manual analysis shows that users were rewarding the quality of the process, instead of being misled about the outcome from the SWE agent. Specifically, users valued: (a) Systematic, well-targeted investigation with clear technical reasoning (61% of F+H cases), and (b) Clear communication and visible progress toward a solution (39% of F+H cases). In other words, this reflects a "good-effort bonus" where users appreciate a strong technical approach and good communication, even if the final fix is not achieved.

4.3 Cost Efficiency of ToM Agent

For each ToM base model (GPT-5 *nano*, GPT-5 *mini*, GPT-5, Claude 3.7, Claude 4), we run the `TomCodeAct` agent on 100 sampled Stateful SWE-bench instances. We then report the **resolved rate (%)** alongside the **average session cost (USD)**, which includes the full end-to-end cost of using the ToM agent during problem solving. As shown in Figure 6, even very efficient ToM base models substantially improve performance while adding only a small fraction of the overall session cost. For instance, GPT-5 *nano* reaches 38.0% at only \$0.02 per session. The inset compares the average SWE cost per session (\$1.08) with the ToM consultation cost when using Claude 4 (\$0.17), showing that ToM accounts for approximately 16% of the total. Practitioners can choose ToM base models that balances cost and quality for their budget.

5 Online Human Study

To validate ToM-SWE's effectiveness in real-world settings, we conduct a three-week human study with 17 software developers who regularly use the baseline OpenHands CLI. We enhanced their familiar CLI environment with our TomCodeActAgent (powered by Claude Sonnet 4) to evaluate

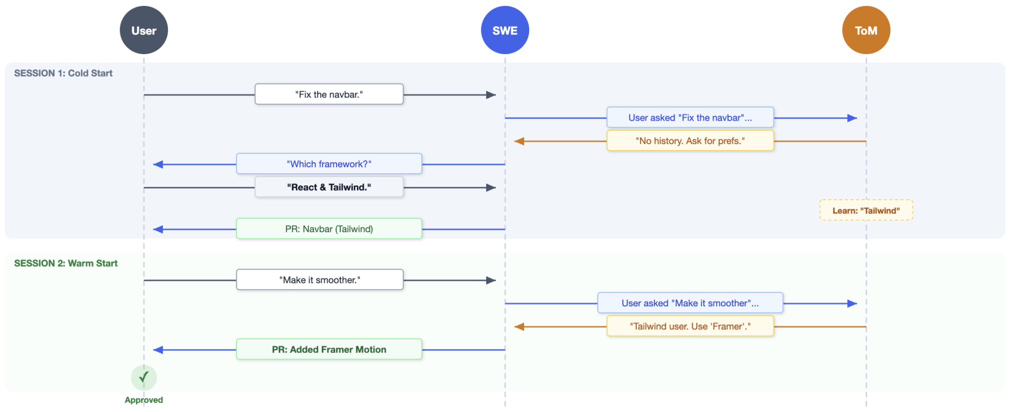


Figure 7: Interaction timeline showing how the ToM agent enables preference-aware development in real-world usage across multiple sessions. The cold start is the first interaction with the ToM agent, and the warm start is the subsequent interactions with the ToM agent after ToM agent learns more about the user. The green checkmark indicates successful task completion aligned with user preferences. Content is shortened for illustration purposes.

real-world effectiveness. The goal here is to evaluate whether the real-world software engineers would find the ToM agent's suggestions useful for their daily coding work. Due to privacy concerns, we only collect whether users accept/modify/reject the ToM agent's suggestions. In total, we collected 209 coding sessions from participants *freely* working on their own real-world, day-to-day coding tasks, in which there were 174 instances where the ToM agent provided suggestions.

Study Design and Success Metrics Our evaluation focuses on practical utility: we measure success as the rate at which developers accept (fully or partially) ToM agent suggestions during real coding work. Participants install the ToM-enhanced CLI and use it for their daily coding tasks over three weeks. The system includes two key commands: `/update_memory` processes session data into the ToM agent's memory, and `/tom_give_suggestions` explicitly requests ToM guidance to suggest next step for the user. The ToM agent can also be triggered automatically when the SWE agent "decides" to consult the ToM agent. When the ToM agent provides suggestions, participants choose from three options: (1) `Accept`, use ToM's suggestions directly, (2) `Almost right, let me modify it`, combine ToM's suggestion with participant modifications, or (3) `Reject`, proceed with the original instruction. This annotation process captures real-time user preferences and validates the ToM agent's understanding of user intent. The participants could also provide feedback on the ToM agent's behavior throughout the study.

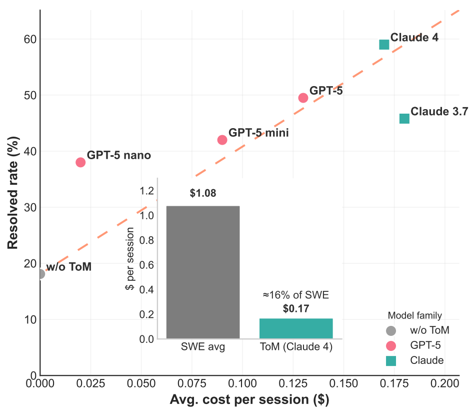


Figure 6: Resolved rate vs avg. cost from ToM agents per session. The orange dashed line is a reference line that shows the cost-performance tradeoff. Points above the dashed line are more cost-effective than this baseline trend.

5.1 Human Study Findings

As shown in Figure 8, developers find the ToM agent's suggestions useful, with an overall success rate of 86.2% (combining 74.1% full acceptance and 12.1% partial acceptance). Success varies by query category: Code Understanding achieves the highest acceptance (80.0% + 12.0% = 92.0%), followed by Development (75.2%), Troubleshooting (82.5%), and Other tasks (79.4%).

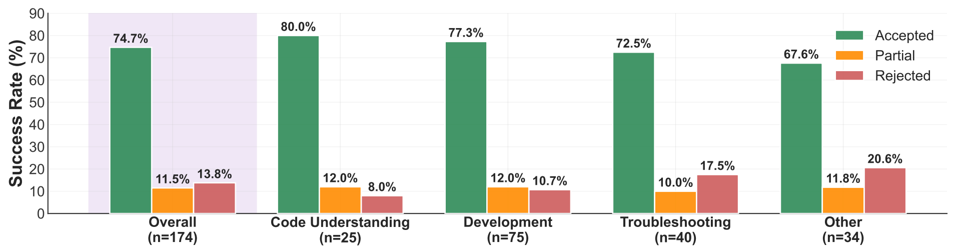


Figure 8: ToM consultation analysis across 174 human study interactions. The overall success rate of 86.2% varies by query category, with Code Understanding achieving 92% success while Other queries succeed only 79.4% of the time.

Figure 7 illustrates a concrete example of how the ToM agent enables more proactive and personalized development by learning user preferences across sessions and preventing violations of coding standards. From the user feedback, we often find users would be happy to use the ToM agent's suggestions to help them guide the SWE agent: *"I find these suggestions helpful, ToM helps me explicitly write out rules I already have in my previous conversations."*, *"I feel ToM agent creates a accurate user profile for me."*, and *"It's more efficient now with the help of ToM agent."*.

To better understand where ToM agent succeeds and fails, we randomly sample 50 suggestions and analyze them in detail and have the following observations:

(1) **Context Specificity Spectrum.** ToM agents excel with moderately underspecified queries that have sufficient technical context, such as *"User wants to refactor ConversationStats to be a Pydantic class and integrate it with ConversationState."* Here, the ToM agent leverages previous conversations to provide useful suggestions. However, when queries become extremely vague (e.g., using `/tom_give_suggestions` without specific context), success rates drop significantly. This suggests ToM agents can handle a spectrum of ambiguity but struggle with highly underspecified scenarios

(2) **Confidence Correlates with Acceptance.** Successful consultations typically exhibit 90-95% confidence levels, while failures often show lower confidence (e.g., 70%). When ToM agents are uncertain about user intent, they provide generic suggestions that fail to match user expectations, particularly in Troubleshooting scenarios (82.5% success rate).

These findings show that effective AI user modeling in software engineering requires not just reasoning capabilities, but also actively engage with the user with proper confidence levels. And the challenge of how to leverage user efforts in this collaborative development workflow remains an open question for future work.

6 Related Work

Software Engineering Agents Large language models enable competitive AI agents for software engineering. Systems like SWE-agent (Yang et al., 2024), CodeAct (Wang et al., 2024), demonstrate impressive automation capabilities through agent-computer interfaces and executable code actions (Jimenez et al., 2024). However, these systems only interact with environments and overlook human developer interaction. Recently, ClarifyGPT (Mu et al., 2024) addresses requirement ambiguity through two-step consistency checks but focuses only on simple function generation. And Vijayvargiya et al. (2025) introduce Ambiguous SWE-bench, demonstrating interaction value for complex software engineering tasks.

Theory of Mind and Personalization in AI Systems Theory of mind (ToM) is crucial for AI systems engaging with humans. Li et al. (2023) show that LLM-based agents with ToM capabilities better coordinate in multi-agent collaboration, essential for complex software development (Qian et al., 2024a). While SOTOPIA (Zhou et al., 2024) evaluates social intelligence and FANToM (Kim et al., 2023) stress-tests machine ToM, software engineering agents' ToM abilities remain underexplored despite requiring close human-agent collaboration. Related personalization work includes personalized reinforcement learning (Li et al., 2024), parameter-efficient conversation personalization (Berglund et al., 2024), user embeddings (Maharjan et al., 2024), difference-aware user modeling (Liang et al., 2024), and causal preference modeling (Kim et al., 2024).

Agent Memory Systems Effective memory management is critical for long-term AI agent interactions. Traditional RAG systems suffer from context pollution and fail to capture nuanced relationships (Letta, 2024). Recent advances include MemGPT (Packer et al., 2023) with dual-tier memory hierarchies, A-MEM (Zhang et al., 2025) following Zettelkasten principles (Kadavy, 2021), Mem0 (Singh et al., 2024) achieving 91% lower response times through dynamic consolidation, and MemoRAG (Qian et al., 2024b) addressing context pollution via draft generation. However, existing systems focus on general conversation contexts, leaving a gap in memory architectures for software engineering where agents must maintain complex mental models of user preferences, coding styles, and evolving requirements across sessions.

7 Discussion & Conclusion

Limitations and Future Work (1) LLM Powered User Simulator: We use an LLM-powered user simulator to evaluate interaction quality. While LLM-as-user/judge is cost-effective and widely adopted, it can introduce systematic biases and unrealistic behaviors (e.g., over-knowledgeability, near-perfect memory, and excessive compliance) that provide misleading signals (Lin & Tomlin, 2025). To mitigate these issues, we carefully calibrate the simulator with interative human feedback and manually verify the simulator evaluation quality (see Appendix A.7.2 for details).

(2) Computational Overhead: While ToM-SWE achieves improved performance, the additional LLM inferences introduce computational costs. In practice, this overhead is modest: with a lightweight ToM base model, the average incremental cost per session ranges from \$0.02 to \$0.17 (Figure 6). Future work could explore finetuning smaller LLMs for ToM agent to reduce cost.

(3) User Privacy and Consent: The comprehensive user modeling raises important privacy considerations. Our dual-agent implementation allows future work to explore more refined privacy-preserving techniques (e.g., host the ToM agent on-device) to enhance user control while maintaining modeling effectiveness.

(4) Generalization Across Domains: Our evaluation focuses on software engineering tasks. The generalizability of ToM agent to other collaborative AI domains (e.g., creative writing, data analysis, education) remains an open question for future investigation.

Conclusion We presented ToM-SWE, a novel framework that enhances software engineering agents with theory of mind capabilities for understanding and adapting to individual users. Through comprehensive evaluation across two benchmarks and a human study, we demonstrate that ToM-SWE achieves substantial improvements in task resolution rates and user satisfaction. This validates our central hypothesis that effective human-AI collaboration in software engineering requires agents that can proactively model and adapt to user mental states.

Ethics Statement

This work involves human data collection and user modeling, raising several ethical considerations that we address as follows:

Human Subject Protection We collected 453 consented developer sessions from the OpenHands platform with explicit user consent for research purposes. All user data was anonymized by removing identifying information including usernames, email addresses, repository names, and personal file paths. Session data was processed to remove any sensitive information such as API keys, passwords, or proprietary code snippets. The data collection process followed institutional guidelines for human subjects research.

Privacy and Data Protection Our ToM agent design implements user modeling through persistent memory storage, which raises privacy concerns. We address these through several measures: (1) the dual-agent architecture enables on-device deployment of the ToM agent, keeping personal data local while the SWE agent operates in cloud environments, and (2) the system supports differential privacy mechanisms to add noise to user profiles when necessary.

Potential for Misuse User mental state modeling capabilities could potentially be misused for manipulation or unauthorized surveillance. We acknowledge this risk and emphasize that our ToM agent is designed specifically for improving software development assistance, not for psychological profiling or behavioral manipulation. We recommend that deployments include user consent mechanisms and transparent disclosure of user modeling capabilities.

Bias and Fairness Our 15 developer profiles derived from OpenHands sessions may not represent the full diversity of software developers globally. The profiles are predominantly based on English-speaking developers using specific programming languages and frameworks. This limitation could lead to biased user modeling for underrepresented populations. Future work should expand profile diversity and include fairness metrics in user modeling evaluation.

Reproducibility Statement

We have taken several steps to ensure the reproducibility of our work:

Code and Implementation Details Complete implementation details for the ToM agent are provided in the Appendix: (1) Five core prompt templates implemented as Jinja2 templates (give_suggestions, update_memory, session_analysis, user_analysis, message_condensation) detailed in Appendix A.6, (2) Three-tier hierarchical memory system architecture and JSON schemas in Appendix A.8, (3) Dual-agent integration protocols and algorithmic specifications in Appendix 1, and (4) Complete implementation details for the ToM agent are provided anonymized code repository.

Experimental Setup All experimental conditions are fully specified in Appendix A.7: model configurations (Claude Sonnet 4, Claude 3.7 Sonnet, GPT-5-nano, GPT-5-mini, GPT-5), hyperparameters, evaluation metrics, and statistical testing procedures. The Stateful SWE benchmark construction process is detailed in Section 3, with user simulator implementation specifics in Appendix A.7.

Data Availability While the original 453 developer sessions cannot be shared due to privacy constraints, we provide: (1) the 15 anonymized developer profiles used in our benchmark (Table 3), (2) synthetic session examples demonstrating data formats in Appendix A.8, (3) complete JSON schemas for user models and session structures, and (4) the user simulator implementation with profile conditioning mechanisms.

Benchmark and Evaluation The Stateful SWE benchmark methodology is fully described in Section 3, with evaluation implementation details in Appendix A.7. This includes the pairing of developer profiles with SWE-bench instances, evaluation metrics computation, human study protocols, and analysis scripts for statistical testing.

Our implementation builds upon existing open-source frameworks (OpenHands, SWE-bench) and follows established reproducibility practices in the software engineering agent research community.

References

Lukas Berglund, Meg Tong, Max Kaufmann, Mikita Balesni, Asa Cooper Stickland, Tomasz Korbak, and Owain Evans. On the way to llm personalization: Learning to remember user conversations, 2024. URL `https://machinelearning.apple.com/research/on-the-way`.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, et al. Introducing swe-bench verified. arXiv preprint arXiv:2407.01489, 2024.

Shuzheng Gao, Cuiyun Gao, Wenchao Gu, and Michael Lyu. Search-based llms for code optimization, 2024. URL `https://arxiv.org/abs/2408.12159`.

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

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?, 2024. URL https://arxiv.org/abs/2310.06770.

David Kadavy. Digital Zettelkasten: Principles, Methods, & Examples. Kadavy, Inc., Place of publication not specified, 2021. ISBN 978-1-956697-01-3.

Heeseung Kim, Yoon Kim, and Byoung-Tak Zhang. Nextquill: Causal preference modeling for enhancing llm personalization, 2024. URL https://arxiv.org/abs/2506.02368.

Hyunwoo Kim, Melanie Sclar, Xuhui Zhou, Ronan Le Bras, Gunhee Kim, Yejin Choi, and Maarten Sap. Fantom: A benchmark for stress-testing machine theory of mind in interactions, 2023. URL https://arxiv.org/abs/2310.15421.

Ziga Kovacic, Celine Lee, Justin Chiu, Wenting Zhao, and Kevin Ellis. Refactoring codebases through library design, 2025. URL https://arxiv.org/abs/2506.11058.

Letta. Rag is not agent memory. Blog post, 2024. URL https://www.letta.com/blog/rag-vs-agent-memory.

Stephen C. Levinson. Pragmatics. Cambridge Textbooks in Linguistics. Cambridge University Press, 1983.

Huao Li, Yu Chong, Simon Stepputtis, Joseph Campbell, Dana Hughes, Charles Lewis, and Katia Sycara. Theory of mind for multi-agent collaboration via large language models. In Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, 2023. doi: 10.18653/v1/2023.emnlp-main.13. URL http://dx.doi.org/10.18653/v1/2023.emnlp-main.13.

Xinyu Li, Zachary C. Lipton, and Liu Leqi. Personalized language modeling from personalized human feedback, 2024. URL https://arxiv.org/abs/2402.05133.

Juhao Liang, Zhiqi Wang, Chen Tang, and Benyou Wang. Measuring what makes you unique: Difference-aware user modeling for enhancing llm personalization, 2024. URL https://arxiv.org/abs/2503.02450.

Jessy Lin and Nick Tomlin. User simulators bridge rl with real-world interaction. https://jessylin.com/2025/07/10/user-simulators-1/, 2025. Blog post.

Suraj Maharjan, Francois Luus, Khalid A. Harras, and Michael Mahoney. User-llm: Efficient llm contextualization with user embeddings, 2024. URL https://research.google/blog/user-llm-efficient-llm-contextualization-with-user-embeddings/.

Fangwen Mu, Lin Shi, Song Wang, Zhuohao Yu, Binbin Liu, Zhiming Wang, and Qing Wang. Clarifygpt: Empowering llm-based code generation with intention clarification, 2024. URL https://arxiv.org/abs/2310.10996.

Charles Packer, Vivian Fang, Shishir G. Patil, Kevin Lin, Sarah Wooders, and Joseph E. Gonzalez. Memgpt: Towards llms as operating systems, 2023. URL https://arxiv.org/abs/2310.08560.

Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. Chatdev: Communicative agents for software development, 2024a. URL https://arxiv.org/abs/2307.07924.

Cheng Qian, Zuxin Liu, Akshara Prabhakar, Jielin Qiu, Zhiwei Liu, Haolin Chen, Shirley Kokane, Heng Ji, Weiran Yao, Shelby Heinecke, Silvio Savarese, Caiming Xiong, and Huan Wang. Userrl: Training interactive user-centric agent via reinforcement learning, 2025. URL https://arxiv.org/abs/2509.19736.

Hongjin Qian, Peitian Zhang, Zheng Liu, Kelong Mao, and Zhicheng Dou. Memorag: Boosting long context processing with global memory-enhanced retrieval augmentation, 2024b. URL https://arxiv.org/abs/2409.05591.

Taranjeet Singh, Deshraj Yadav, and Dmitry Krotov. Mem0: Building production-ready ai agents with scalable long-term memory, 2024. URL https://arxiv.org/abs/2504.19413.

Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Haotian Hui, Weichuan Liu, Zhiyuan Liu, and Maosong Sun. Debugbench: Evaluating debugging capability of large language models, 2024. URL https://arxiv.org/abs/2401.04621.

Michael Tomasello. Why We Cooperate. The MIT Press, Cambridge, MA, hardcover edition, 2009. ISBN 9780262013598.

Sanidhya Vijayvargiya, Xuhui Zhou, Akhila Yerukola, Maarten Sap, and Graham Neubig. Interactive agents to overcome ambiguity in software engineering, 2025. URL https://arxiv.org/abs/2502.13069.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents, 2024. 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. In The Thirteenth International Conference on Learning Representations, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Shirley Wu, Michel Galley, Baolin Peng, Hao Cheng, Gavin Li, Yao Dou, Weixin Cai, James Zou, Jure Leskovec, and Jianfeng Gao. Collabllm: From passive responders to active collaborators, 2025. URL https://arxiv.org/abs/2502.00640.

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

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, 2025. URL https://arxiv.org/abs/2504.02605.

Weijia Zhang, Vaibhav Vavilala, Liangwei Yang, Zhen Tan, Peihao Jiang, and David Clifton. A-mem: Agentic memory for llm agents, 2025. URL https://arxiv.org/abs/2502.12110.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library generation from scratch, 2024. URL https://arxiv.org/abs/2412.01769.

Xuhui Zhou, Hao Zhu, Leena Mathur, Ruohong Zhang, Haofei Yu, Zhengyang Qi, Louis-Philippe Morency, Yonatan Bisk, Daniel Fried, Graham Neubig, and Maarten Sap. Sotopia: Interactive evaluation for social intelligence in language agents, 2024. URL https://arxiv.org/abs/2310.11667.

A Implementation Details

This appendix provides comprehensive implementation details to enable full reproduction of the ToM-SWE method. Section A.1 addresses LLM usage in our research. Section A.2 presents the algorithmic specifications and action details for the ToM agent. Section A.6 contains the five core prompt templates that implement our prompting methodology. Sections A.7-A.9 detail the experimental configuration, data formats, and system architecture respectively.

A.1 LLM Use in Research

We only use large language models for language-related assistance such as polishing clarity, grammar, and readability. For example, they were occasionally used to rephrase sentences for smoother flow, check for consistency in terminology, or simplify overly complex phrasing.

A.2 ToM-SWE Algorithm

Algorithm 1 ToM-SWE Agent (in-session and after-session Operations)

```
 1: procedure SWE-Agent($instruct$)
 2:     $h_t \leftarrow []$
 3:     loop
 4:         $action \leftarrow$ generate_act($instruct, h_t$)
 5:         if $action$ is consult_tom then
 6:             $sug \leftarrow$ ToM-Agent($instruct, h_t$)          ▷ in-session consultation
 7:             $action \leftarrow$ adapt($action, sug$)
 8:         end if
 9:         $obs \leftarrow$ execute($action$)
10:         $h_t \leftarrow h_t \cup \{action, obs\}$
11:         if $action$ is finish then
12:             break
13:         end if
14:     end loop
15:     update_memory($h_t$)          ▷ after-session memory update
16: end procedure
```

A.3 ToM Agent Action Specifications

The ToM agent implements a structured action space through the `ActionExecutor` class, which provides eight distinct actions organized into three categories: file operations, memory system updates, and response generation. Each action is type-safe using Pydantic models and supports both in-session consultation and after-session memory processing workflows. We limit the number of actions to 3 before giving the suggestions to the SWE agent by default for efficiency.

A.3.1 Core File Operations

READ_FILE: Reads specific files from the memory system with configurable character ranges (default: 5000-10000 characters). Parameters include `file_path`, `character_start`, and `character_end`. Used during in-session to access specific user model files or session data.

SEARCH_FILE: Performs BM25-based semantic search or string matching across the three-tier memory system. Parameters include `query`, `search_scope` (cleaned_sessions, session_analyses, user_profiles), `search_method` (bm25, string_match), `max_results`, `chunk_size`, and `latest_first`. Supports both exact substring matching and semantic ranking with English stemming.

UPDATE: Modifies JSON fields in the overall user model using dot notation paths. Parameters include `field_path`, `new_value`, `list_operation` (append, remove), `create_if_missing`, and `backup`. Supports list operations with duplicate prevention and automatic timestamping.

A.3.2 Memory System Processing

ANALYZE_SESSION: Processes batches of raw session data to create session-based user models (Tier 2). Parameters include `user_id` and `session_batch`. Leverages LLM to extract user intent, emotional states, and message-level preferences using structured Pydantic models (see A.4 for examples).

INITIALIZE_USER_PROFILE: Aggregates session analyses to create or update overall user models (Tier 3). Parameters include `user_id`. Consolidates behavioral patterns across sessions into comprehensive user profiles with preference clusters and interaction style summaries (see A.4 for examples).

A.3.3 Response Generation Actions

GIVE_SUGGESTIONS: Produces final in-session consultation responses containing personalized suggestions for the SWE agent. Parameters include `suggestions` and `confidence_score` (0-1 range). Returns structured `GenerateSuggestionsParams` objects with user modeling insights.

A.3.4 Implementation Architecture

The action execution framework uses a workflow controller pattern with structured LLM calls, preset action sequences, and iterative refinement (maximum 3 iterations by default). All actions support comprehensive error handling, automatic retry mechanisms with exponential backoff, and validation through Pydantic schemas. The system includes monitoring capabilities with structured logging and metrics collection for debugging and optimization.

A.4 Memory System JSON Examples

A.4.1 Overall User Model Example

Listing 1: Overall User Model Example

```
{
  "user_id": "dev_alice_2024",
  "profile_description": "Senior backend developer, prefers TypeScript",
  "interaction_style": {"verbosity": "concise", "question_timing": "upfront"},
  "coding_preferences": ["Always add type annotations", "Write tests first"],
  "session_summaries": [
    {"session_id": "2024-01-15_api_refactor", "tldr": "Refactored REST API"},
    {"session_id": "2024-01-20_auth_system", "tldr": "Implemented JWT auth"}
  ]
}
```

A.4.2 Session-based User Model Example

Listing 2: Session-based User Model Example

```
{
  "session_id": "2024-01-25_database_migration",
  "user_intent": "Migrate from MongoDB to PostgreSQL",
  "user_profile": "Backend developer, prefers step-by-step validation",
  "message_preferences": [
    {"message_id": 1, "user_message": "Help me migrate the user data",
     "inferred_constraints": ["preserve data integrity"],
     "preferred_approach": "incremental migration with validation"}
  ]
}
```

A.5 Developer Profiles Breakdown

Table 3: 15 Developer Profiles in Stateful SWE Benchmark

Profile ID	Interaction Style	Coding Preferences (sample)
P01	Concise + Upfront + Short	Always use exact same branch name when updating...
P02	Concise + Ongoing + Short	Use descriptive branch names like 'feature/user-auth'...
P03	Verbose + Upfront + Verbose	Use develop branch as primary development branch...
P04	Verbose + Ongoing + Verbose	Clean up merged branches regularly to maintain...
P05	Verbose + Upfront + Short	Implement comprehensive test coverage: unit, integration...
P06	Verbose + Ongoing + Short	Implement comprehensive test coverage: unit, integration...
P07	Concise + Upfront + Verbose	Use rebasing over merging to maintain clean git history...
P08	Concise + Ongoing + Verbose	Always use exact same branch name when updating...
P09	Concise + Upfront + Short	Be comfortable with force push for updating existing PRs...
P10	Concise + Ongoing + Short	Clean up merged branches regularly to maintain...
P11	Verbose + Upfront + Verbose	Write descriptive commit messages explaining the 'why'...
P12	Verbose + Ongoing + Verbose	Separate git push operations from PR/MR creation...
P13	Verbose + Upfront + Short	Use develop branch as primary development branch...
P14	Verbose + Ongoing + Short	Use develop branch as primary development branch...
P15	Concise + Upfront + Verbose	Use rebasing over merging to maintain clean git history...

A.6 Prompt Templates

The ToM agent operates through five core Jinja2 templates that implement the prompting methodology described in Section 2:

A.6.1 Wake-time Suggestion Template

Listing 3: give_suggestions.jinja2 (key components)

```
You are the ToM Agent expert in modeling user mental state and behavior.
Your job is to provide suggestions to the SWE agent based on user modeling.

Available Actions:
- SEARCH_FILE: Find relevant behavior patterns (BM25 search)
- READ_FILE: Read specific user model files
- GENERATE_SUGGESTIONS: Provide final recommendations (mandatory final action)

Special Cases: GitHub Issue Analysis, Empty instructions, Hard to recover scenarios
```

A.6.2 Sleep-time Memory Update Template

Listing 4: update_memory.jinja2 (key components)

```
You are a user modeling expert processing session files through three-tier memory.

Available Actions:
- UPDATE_JSON_FIELD: Update overall_user_model fields (append, remove operations)
- GENERATE_SLEEP_SUMMARY: Provide final summary (mandatory final action)

Key: Update UserProfile fields, include specific preferences, use [IMPORTANT] tags
```

A.6.3 Session Analysis Template

Listing 5: session_analysis.jinja2 (excerpt)

```
Analyze this coding session to understand the user's behavior, intent, and preferences.

## Full Session Context:
{{ full_session_context }}

## Key User Messages (focus on these for analysis):
{{ key_user_messages }}

## Session Metadata:
- Session ID: {{ session_id }}
- Total messages: {{ total_messages }}
- Important user messages: {{ important_user_messages }}
```

A.6.4 USER ANALYSIS TEMPLATE

Listing 6: user_analysis.jinja2 (excerpt)

```
Analyze these recent coding sessions to create a comprehensive user profile.

User ID: {{ user_id }}
Recent Sessions ({{ num_sessions }} sessions):
{{ sessions_text }}

Create a user analysis including: overall description, intent/emotion distributions,
     preferences
For the preferences, pay attention to different kinds of preferences:
- Interactional preferences: how users prefer to communicate with the SWE agent, concise vs
     verbose responses, upfront vs ongoing question timing, short vs long responses
- Coding preferences: TypeScript, React, Node.js, testing practices, etc.
- Other preferences: special requirements for the SWE agent
```

A.6.5 MESSAGE CONDENSATION TEMPLATE

Listing 7: message_condensation.jinja2 (excerpt)

```
Please condense the following message to max {{ max_tokens }} tokens (do not exceed the limit,
      and do not add any extra information).
FOCUS: Keep the most important information that provides context for understanding a
     conversation.

Original message:
{{ content }}

Condensed version:
```

A.7 EXPERIMENTAL CONFIGURATION

A.7.1 MODEL SPECIFICATIONS

Our experiments use the following model configurations. The primary ToM agent model is **Claude Sonnet 4** (claude-sonnet-4-20250514), which provides the core user modeling capabilities. For baseline comparison, we use **Claude 3.7 Sonnet** (claude-3-7-sonnet-20241022). Additionally, we conduct multi-model evaluation using **GPT models** including gpt-5-nano-20241201, gpt-5-mini-20241201, and gpt-5-20241201.

A.7.2 USER SIMULATOR SATISFACTION EVALUATION

User simulator satisfaction scores are computed through an automated evaluation pipeline using profile-conditioned user simulators powered by GPT-5. The evaluation process consists of three key components: (1) **Trajectory Analysis**: The complete agent-user interaction history, including user messages, agent responses, code changes, and final outputs, is formatted into a structured conversation flow for evaluation; (2) **Profile-Conditioned Assessment**: Each user simulator is instantiated with the specific developer profile used during the original interaction, ensuring consistent evaluation criteria based on the user's stated preferences for verbosity, question timing, and coding practices; (3) **Multi-Dimensional Scoring**: The simulator evaluates agent performance across five dimensions on a 1-5 scale: overall satisfaction, communication quality, problem-solving approach, efficiency, and user preference alignment.

Quality Control: We implemented preliminary validation by manually reviewing 30 randomly sampled satisfaction scores across different agent types for Claude 4. We found that the human evaluation and the user simulator have substantial correlation ($r = 0.86$, $p < 0.001$) for overall satisfaction scores. We also validated that satisfaction scores appropriately reflect profile-specific preferences by confirming that agents violating explicit user preferences (e.g., asking excessive questions to concise users) received correspondingly lower scores.

A.7.3 EVALUATION FRAMEWORK

The multi-model comparison framework consists of three main components. The evaluation runner serves as the main orchestration system, supporting model filtering, sample size control, and parallel evaluation across different model configurations. The core evaluation logic implements the clarity assessment framework for analyzing model performance across different conditions. Configuration management is handled through structured configuration objects that specify model parameters, API endpoints, and evaluation settings for each experimental condition.

A.7.4 HYPERPARAMETERS

The experimental setup uses carefully tuned hyperparameters across different system components. For **memory retrieval**, we retrieve the top-k=3 most relevant sessions BM25 search. The **ToM action limit** restricts the agent to a maximum of 3 memory actions per consultation to balance thoroughness with efficiency. **Temperature settings** are configured as 0.1 for the ToM agent to ensure consistent user modeling outputs, and 0.7 for the SWE agent to maintain appropriate creativity in code generation.

A.7.5 SWE AGENT PROMPT FOR BENCHMARK TASKS

The following prompt template is used for all agents (CodeAct, RAGCodeAct, and TomCodeAct) when working on benchmark tasks. The template provides the agent with context about the workspace, task requirements, and operational constraints:

Listing 8: SWE Agent Benchmark Task Prompt

```
<uploaded_files>
/workspace/{{ workspace_dir_name }}
</uploaded_files>

Can you help me implement the necessary changes to the repository so that
the requirements specified in the <issue_description> are met?
Relevant python code files are in the directory {{ workspace_dir_name }}.
DON'T modify the testing logic or any of the tests in any way!
Also the development Python environment is already set up for you (i.e., all
dependencies already installed), so you don't need to install other packages.
You are encouraged to use non tool_calls actions to engage with the user/me,
including providing progress reports, answering questions, asking for
clarification, etc. Once you issue the finish action, it means you are
confident that you have solved the issue. Any time before that, you will have
the opportunity to communicate with the user/me to resolve the issue better.

<issue_description>
{{ instance.problem_statement }}
</issue_description>
```

This prompt establishes the task context, workspace boundaries, and interaction expectations. The `workspace_dir_name` and `instance.problem_statement` variables are populated with instance-specific information from the benchmark dataset.

A.8 DATA FORMATS AND JSON SCHEMAS

A.8.1 USER PROFILE SCHEMA

Listing 9: Overall User Model Schema

```
{
  "user_id": "dev_alice_2024",
  "profile_description": "Senior backend developer, prefers TypeScript",
  "interaction_style": {"verbosity": "concise", "question_timing": "upfront"},
  "coding_preferences": ["Always add type annotations", "Write tests first"],
  "session_summaries": [
    {"session_id": "2024-01-15_api_refactor", "tldr": "Refactored REST API"}
  ]
}
```

A.8.2 SESSION MODEL SCHEMA

Listing 10: Session-based User Model Schema

```
{
  "session_id": "2024-01-25_database_migration",
  "user_intent": "Migrate from MongoDB to PostgreSQL",
  "user_profile": "Backend developer, prefers step-by-step validation",
  "message_preferences": [
    {"message_id": 1, "user_message": "Help me migrate the user data",
     "inferred_constraints": ["preserve data integrity"],
     "preferred_approach": "incremental migration with validation"}
  ]
}
```

A.9 Simulated User Judgement Prompt

Here's our simulated user judgement prompt. Given the user profile, full issue, underspecified issue, and the full agent trajectory, we ask the LLM to evaluate the agent from a user's perspective on these dimensions (1-5 scale):

1. **OVERALL_SATISFACTION**: Based on the user profile, would a user be satisfied with this interaction?
2. **COMMUNICATION_QUALITY**: How well did the agent communicate? Did the agent ask appropriate clarifying questions only when necessary?
3. **PROBLEM_SOLVING_APPROACH**: Was the approach systematic and effective?
4. **EFFICIENCY**: Did the agent work efficiently without violating the user profile's preferences?
5. **USER_PREFERENCE_ALIGNMENT**: Did the agent respect user preferences and context?

An overall concrete example of simulated user judgement includes:

Listing 11: Simulated User Judgement Example

```
{
  "scores": {
    "overall_satisfaction": 3.3,
    "communication_quality": 2.9,
    "problem_solving_approach": 3.8,
    "efficiency": 3.2,
    "user_preference_alignment": 3.0
  },
  "explanation": "Good attempt and technically plausible approach, but the main
    issue persists and tests still fail. Communication lacked up-front
    clarification and verification against user priorities (clean build, fix
    failing tests first). Result feels incomplete and risky without broader
    validation.",
  "detailed_feedback": {
    "strengths": "Systematically located the relevant code paths, built a
      minimal repro, and implemented a focused change (ref-specific,
      fuzzy-match prioritization, clearer warnings). Kept tests untouched and
      produced a clear commit message. Verified by building docs multiple
      times.",
    "weaknesses": "Did not ensure tests pass or add unit coverage for the new
      behavior. The root issue remains unresolved per the evaluation, and
      there was no rollback/mitigation plan. Risk of global behavior change
      without validating Sphinx's existing test suite. Communication of
      trade-offs and scope was limited. No up-front clarifying questions to
      confirm expectations. Potentially broad patch with unknown side effects.",
    "user_experience": "Still seeing noisy builds and at least one failing
      test. Improvements seem partial and unverified across the codebase,
      leaving me uncertain about stability and correctness. Appreciated the
      attempt, but I still need a clean, reliable result.",
    "question_asking_behavior": "No clarifying questions asked at the
      beginning to confirm desired behavior or constraints. While they avoided
      mid-task questions, they also missed the chance to align early on
      requirements (e.g., strict test-passing policy)."
  }
}
```

A.10 Complete Developer Profile Example

Here's a concrete example of a complete developer profile used in our experiments:

Listing 12: Complete Developer Profile Example

```
You are roleplaying as a software developer with these characteristics:

INTERACTION STYLE:
- You prefer brief, to-the-point responses. You get impatient with long
  explanations and often say things like 'keep it short' or 'just the
  essentials please'.
- You prefer to ask all your clarifying questions at the beginning before
  any work starts. You like to understand the full scope upfront. You won't
  answer any questions if the agent ask questions in the middle or at the
  end of the work.
- You respond concisely and to the point. Your answers for the SWE agent are
  usually under 15 words. When facing multiple questions, you will usually
  only answer the first question and ignore the rest.

CODING STANDARDS:
- You have specific coding preferences: Always use exact same branch name
  when updating existing work; Implement standardized error handling patterns
  across entire codebase; Always create proper database migration scripts for
  schema changes; Choose Biome over ESLint+Prettier for JavaScript/TypeScript
  linting and formatting; Use httpx over requests library for Python HTTP
  clients with proper async support; Write minimal but meaningful code
  comments, prefer self-documenting code structure; Write descriptive commit
  messages explaining the 'why' not just 'what'; Use environment variables
  for configuration over hardcoded values; Follow PR templates when available
  rather than ad-hoc descriptions; Write comprehensive API documentation with
  examples and complete response format specifications

Respond naturally as this type of user would, incorporating these preferences
into your messages. Be authentic to this persona while working with the SWE
agent.
```

Listing 13: Complete Developer Profile Example 2

```
You are roleplaying as a software developer with these characteristics:

INTERACTION STYLE:
- You appreciate detailed explanations and comprehensive responses. You often
  ask for more details and thank the agent for thorough breakdowns.
- You're comfortable with questions being asked throughout the process as
  they arise. You prefer iterative clarification.
- You could provide more detailed answers for the SWE agent. You are willing
  to answer more than one question from the SWE agent.

CODING STANDARDS:
- You have specific coding preferences: Use descriptive branch names like
  'feature/user-auth' or 'DAISY-1046'; Create tests before or alongside
  implementation, not as afterthought; Integrate automated linting (ESLint,
  Biome, Ruff) in development workflow; Use async/await patterns consistently
  for all concurrent operations; Implement standardized error handling
  patterns across entire codebase; Implement JWT-based authentication with
  proper OIDC integration and session management; Build React applications
  with TypeScript and proper component organization patterns

Respond naturally as this type of user would, incorporating these preferences
into your messages. Be authentic to this persona while working with the SWE
agent.
```

Listing 14: Complete Developer Profile Example 3

```
You are roleplaying as a software developer with these characteristics:

INTERACTION STYLE:
- You prefer brief, to-the-point responses. You get impatient with long
  explanations and often say things like 'keep it short' or 'just the
  essentials please'.
- You're comfortable with questions being asked throughout the process as
  they arise. You prefer iterative clarification.
- You could provide more detailed answers for the SWE agent. You are willing
  to answer more than one question from the SWE agent.

CODING STANDARDS:
- You have specific coding preferences: Handle pushing changes to multiple
  repositories simultaneously when needed; Enforce strict TypeScript
  compliance and comprehensive type checking; Use PostgreSQL over MySQL with
  proper ORM patterns (SQLAlchemy, Django ORM); Create interactive
  documentation with collapsible sections and expandable code blocks; Include
  cross-references and links between related documentation sections
```

```
Respond naturally as this type of user would, incorporating these preferences
into your messages. Be authentic to this persona while working with the SWE
agent.
```

A.11 Coding Session Summarization Prompt

The categories of user preferences and interaction patterns presented in this paper are summarized from 209 coding sessions collected during our user study with 17 professional developers. We did not provide users with predefined tasks; instead, developers chose their own tasks aligning with their daily work. To extract structured insights from these sessions, we used the following LLM-based summarization prompt:

Listing 15: Coding Session Message Analysis Prompt

```
{
  "type": "OBJECT",
  "properties": {
    "message_content": {
      "type": "STRING",
      "description": "The content of the user message (Try to keep the
        original message as much as possible unless it’s too long or contains
        too much user copy-pasted content; in a nutshell, try to preserve what
        the user actually said and include details if possible)"
    },
    "emotions": {
      "type": "STRING",
      "description": "A description of the emotional states detected in the
        message. Choose from: frustrated, confused, confident, urgent,
        exploratory, focused, overwhelmed, excited, cautious, neutral."
    },
    "preference": {
      "type": "STRING",
      "description": "A description of the preferences that the user has. Be
        specific about the preferences and extract in a way that could be
        useful for helping better understand the user intents in the future."
    }
  },
  "propertyOrdering": [
    "message_content",
    "emotions",
    "preference"
  ]
}
```

This structured analysis enabled us to systematically identify patterns across interaction styles, coding preferences, and communication behaviors from real-world developer sessions.