Grounding LLM Agent Intent in Multi-Module Environments with Executable Action Programs

Anonymous ACL submission

Abstract

Large Language Model (LLM) agents are increasingly expected to interact with external environments rather than only generate text. A key challenge in this process is to accurately and efficiently ground high-level agent intents into valid executable action sequences, especially when the environment consists of multiple interacting modules. To address this challenge, we introduce *executable action programs* (EAPs), a restricted Python-based intermediate representation between textual agent intents and structured environment-module action calls. Due to the expressive power of Python code, EAPs can explicitly represent complex behavioral sequences, their branching relationships, and cross-module data dependencies. Based on this representation, we implement **EAPRouter**, which generates, validates, executes, and reuses EAPs through an agent-EAP interface and a lightweight environment module integration protocol. We evaluate EAPRouter on real agent-environment interaction traces from social simulation, a demanding application scenario with multi-agent, multi-module, and repeated interaction patterns. Experiments show that EAPRouter achieves a 60.0% grounding success rate with Kimi-K2-Instruct, outperforming the SOTA function-calling baseline by 31.6%. It also improves over the strongest baselines by 11.5%-25.4% across GLM-4.7, Claude Opus 4.5, and Gemini 2.5 Flash. In terms of efficiency, EAP pre-generation handles more than 70% of interaction requests, cache reuse handles 66.5% of the remaining requests without new generation calls. Its safety validation also blocks all tested unsafe payloads and prompt-injection attempts. The code is available at `https://anonymous.4open.science/r/EAPRouter-E7BD`.

1 Introduction

With the rapid development of large language model (LLM) agents (Park et al., 2023; Gao et al., 2024; Piao et al., 2025; Yang et al., 2024b), LLM agents are increasingly expected to interact with external environments to execute actions and obtain feedback. In this interaction process, a key step is to ground the agent's high-level intent into a valid action sequence over the environment. At present, the dominant technique for realizing this grounding process is function calling (Wang et al., 2025b; Patil et al., 2025), where an LLM outputs a structured text containing the target function name and argument values, and a handler executes the corresponding function call. This technique has achieved success in application scenarios such as programming and software engineering (Jimenez et al., 2024; Yang et al., 2024a), web navigation (Deng et al., 2023; Zhou et al., 2024), embodied interaction (Shridhar et al., 2021; Xu et al., 2023), and general tool-use evaluation (Patil et al., 2025).

However, in complex application scenarios such as LLM agent based social simulation (Park et al., 2023; Piao et al., 2025; Zhang et al., 2025; Yang et al., 2024b; Wang et al., 2025a), the complexity of the environment and the efficiency requirements for agent interaction are both substantially increased. The environment changes from a single group of action interfaces to a combination of multiple environment modules, and the interaction scale changes from a single agent to hundreds or thousands of agents. Such scenarios usually contain heterogeneous simulation modules and require agents to understand and interact with them effectively. Facing this type of setting, existing function-calling techniques reveal clear limitations. On the one hand, during the grounding of agent intent, coupling and dependencies among environment modules require the agent to understand cross-module dependencies and remember previous action sequences in order to invoke the next action correctly, which places a high demand on the agent's long-term processing ability. On the other hand, function calling tends to produce multiple invocation requests that are

difficult to reuse, because it usually specifies concrete argument values and follows a step-by-step invocation pattern. This makes it difficult to handle large-scale agent interaction scenarios efficiently. Overall, grounding LLM agent intent into complex multi-module environments urgently requires an accurate and efficient method.

To address this problem, we propose *executable action programs* (EAPs) as an intermediate representation between LLM agent intents and multi-module environment actions. EAPs borrow the expressive power of Python-like syntax and the rich ecosystem of packages to represent environment action sequences, branch conditions, parameter parsing and transformation rules, and result aggregation procedures in a reusable way. As a result, EAPs can support accurate and efficient agent intent grounding. To adapt to this intermediate representation, we further design a standardized agent-EAP interface that supports bidirectional transfer of both textual intents and structured data. We also standardize the EAP-environment-module interface so that existing environment modules can be rapidly integrated. In implementation, we design **EAPRouter** to generate EAPs, reuse them through caching based on intent and input-structure similarity, conduct safety checks, and execute them.

To evaluate the accuracy and efficiency of EAPs, we conduct extensive experiments. By integrating 15 environment modules and constructing a benchmark from real agent-environment interaction traces, experiments demonstrate that EAPRouter achieves an approximately 60% grounding success rate for agent intents in complex application scenarios, outperforming function-calling baselines by 25.4%-31.6% with the strongest LLMs. Long-horizon interaction experiments with 100 agents further show that EAP pre-generation and caching substantially reduce the extra overhead of intent grounding: pre-generated special EAPs handle more than 70% of interaction requests, and cache reuse handles 66.5% of the remaining requests without new generation calls. These results verify that EAPs can accurately and efficiently ground agent intents in complex multi-module environments. In addition, to test the EAPRouter's resistance to attacks, we conducted safety validation experiments. The results showed that the EAPRouter's security mechanisms were able to completely block all malicious payloads and prompt-injection attempts.

2 Related Works

2.1 LLM Agents and Environment Interaction

LLM agents interact with external environments through different forms of action interfaces. Early environment-interaction benchmarks usually define semi-structured textual actions. For example, ALFWorld (Shridhar et al., 2021) connects text-based decision making with embodied household environments, where an agent issues textual commands and receives environment feedback. Urban embodied environments (Xu et al., 2023) similarly require agents to understand environment states and produce textual actions under specific protocols. These settings demonstrate the value of environment feedback for grounding agent behavior, but their action spaces are usually designed for specific environments and are difficult to directly generalize to heterogeneous multi-module systems.

Function calling provides a more standardized way to connect LLM agents with external tools and environment APIs (Wang et al., 2025b; Patil et al., 2025). By asking the LLM to output a target function name and structured arguments, function calling makes tool invocation easier to parse and execute. It has also become a common basis for agent patterns such as reasoning-action loops (Yao et al., 2022) and plan-and-execute methods (Wang et al., 2023). However, function calling typically grounds an intent through step-by-step invocation. When a single agent intent requires multiple environment modules, later calls may depend on observations returned by earlier calls, and such dependencies must be maintained implicitly across the interaction history. Moreover, because each call usually contains concrete argument values, the resulting invocation traces are difficult to reuse across agents with similar intents but different contexts.

Recent work also explores using generated programs as intermediaries for repetitive requests. GenCache (Chakraborty et al., 2025) shows that generated code can be reused to handle repetitive tasks more efficiently than repeated inference, suggesting that executable programs can effectively mediate large-scale similar requests. However, GenCache mainly focuses on reusable computation for repeated tasks, rather than the interaction process between LLM agents and complex multi-module environments. Therefore, while this work suggests that the EAP method is a viable approach to addressing the grounding of agent intentions, ef-

fective method design and implementation are still needed to make it a reality.

2.2 LLM Code Generation

In recent years, the code generation capabilities of LLMs have improved substantially. Current mainstream LLMs, such as GLM-4.7[1], Claude Code 4.5[2], and Qwen2.5-Coder (Hui et al., 2024), can handle many common coding tasks and have been evaluated on challenging benchmarks such as LiveCodeBench (Jain et al., 2024) and SWE-bench (Jimenez et al., 2024). These advances make it practical for LLMs to generate executable programs with function calls, variable bindings, and complex control logic. Given these recent advancements in LLM-generated code, we believe that the EAP-based agent intent grounding method is highly feasible.

3 Method

3.1 Overview

To address the challenge of accurately and efficiently grounding LLM agent intents in multi-module environments, we propose an executable action program (EAP) based method, as shown in Figure 1. This method introduces EAPs as an intermediate representation between textual agent intents and structured environment-module action calls. Concretely, an EAP is instantiated as a restricted Python program. It keeps the expressive power of executable programs for composing actions, while constraining the syntax, built-in functions, inputs, and outputs for safe and predictable environment interaction. With this representation, a complex grounding process that would otherwise require multi-turn LLM-based function calling and implicit intermediate-variable management can be converted into a reusable executable program generated by a single LLM call.

To make EAPs compatible with different agent designs, we first define an agent-EAP interface whose inputs and outputs both contain textual instructions and structured data. To make existing environment modules callable from EAPs, we further design a concise environment module integration protocol that standardizes the form of callable environment actions.

Based on the above representation and interfaces, we implement **EAPRouter**, an LLM-driven EAP generator and executor. EAPRouter extracts and aggregates action interfaces from multiple environment modules during initialization and pre-generates EAPs for frequent requests. At runtime, it supports LLM-driven EAP generation, EAP caching based on instruction-template semantics and structured-data similarity, EAP safety validation, and EAP execution. Through EAPRouter, we realize accurate and efficient grounding of LLM agent intents in multi-module environments for large-scale agent interaction.

3.2 Executable Action Programs

In our design, an executable action program is a Python code snippet with restricted syntax, restricted built-in functions, and fixed input-output conventions. The input of an EAP contains the instruction from the agent and structured data named `context`. All environment modules are injected into the EAP execution context as a collection named `modules`, so that the EAP can invoke environment module actions through their Python interfaces. The structured output is required to be written into a designated variable named `result`. EAPs are also allowed to output textual information through `print()`, and these messages are captured for generating textual feedback to the LLM agent.

Listing 1: An example executable action program.

```
instruction = (
    "Go to Blue Maple Bistro (POI ID:
    {poi_id}) "
    "and start a 30-minute eating out event."
)
context = {"id": 3, "variables": {"poi_id":
    700002365}}

# ==== EAP BEGIN ====

MobilitySpace = modules["MobilitySpace"]
EventSpace = modules["EventSpace"]
person_id = context["id"]
poi_id = context["variables"]["poi_id"]

poi = await MobilitySpace.get_poi(
    poi_id=poi_id
)
await MobilitySpace.move_to(
    person_id=person_id,
    aoi_id_or_poi_id=poi_id,
    mode="driving"
)
person = await MobilitySpace.get_person(
    person_id=person_id
)
event = await EventSpace.start_event(
    person_id=person_id,
    event_type="eating out",
    event_name=poi.name,
    expected_duration_seconds=1800
```

[1] https://z.ai/blog/glm-4.7

[2] https://claude.com/product/overview

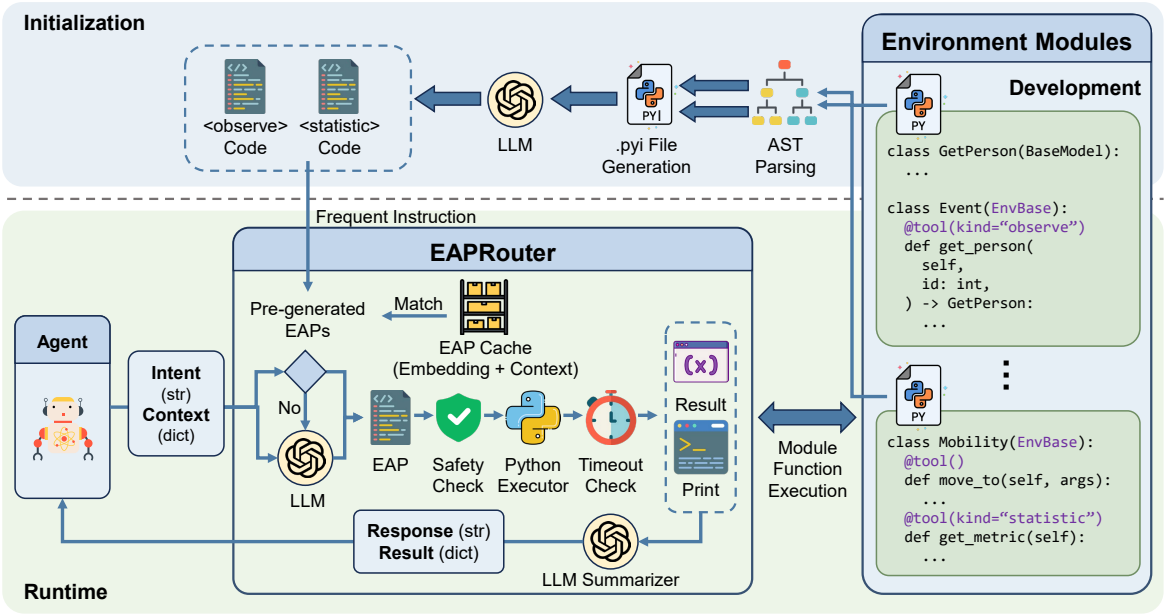

Figure 1: Overview of our EAP-based method for grounding LLM agent intents in multi-module environments.

```
)
print(f"Agent {person_id} started
    {event.name}.")
result = {"person": person, "event": event}

# ==== EAP END ====
```

Listing 1 shows a simplified EAP example adapted from the complete runtime process in Appendix C. The agent intends to go to a restaurant and start a 30-minute eating-out event. The EAP receives a textual instruction and a structured `context`. The instruction describes the target behavior, while `context` provides the precise agent ID and POI ID used for argument grounding. The program then retrieves the required modules from `modules`, calls `MobilitySpace` to query the POI, move the agent, and read the updated person state, and then calls `EventSpace` to start the eating-out event. Finally, it writes structured execution results into `result` and uses `print()` to provide textual feedback.

This design provides two explicit channels for textual and structured information, making EAPs general and compatible with different agent designs. At the same time, EAPs can use variables, branches, loops, and other Python constructs to express cross-module dependencies, action ordering, and other low-level execution logic behind agent intents. Finally, the same EAP can receive different `context` values and parse different argument values from them. This makes EAP reuse possible and reduces repeated LLM calls in large-scale scenarios where many agents issue structurally similar interaction requests.

3.3 Agent-EAP Interface

Based on the EAP design above, the interface between agents and EAPs is clear. EAPs receive textual instructions and structured `context` from agents, and return textual feedback captured from `print()` together with structured data stored in `result`. This input-output convention allows EAPs to serve as a stable grounding layer between flexible agent intents and concrete environment action calls.

On the input side, textual instructions, including natural-language commands or template strings, allow agents to express combined or fuzzy commands over multiple environment modules. Structured `context` provides precise grounding variables such as agent IDs, event durations, and template values. Therefore, agents do not need to encode all parameters into free-form text, and EAPs can extract arguments from a structured source when invoking environment actions. On the output side, environment interaction results are returned both as textual responses and as structured data, supporting downstream agent processing such as memory updates (Xu et al., 2025; Chhikara et al., 2025). Through this design, the agent is decoupled from specific environment implementations while the flexibility and integrity of information transfer

are preserved.

3.4 Environment Module Integration

To make EAPs executable over existing environment modules, we design a simple and easy-to-use environment module integration protocol. Similar to lightweight tool frameworks such as fastmcp[3], developers only need to make existing environment code inherit from the `EnvBase` base class and add the `@tool()` Python decorator to functions that can be called by EAPs. The decorator also supports an optional `kind` parameter, such as `"observe"` or `"statistic"`, to mark actions used by frequent requests. This flag will be used by EAPRouter to pre-generate EAPs for common agent-environment interactions, as described in the next subsection. Type hints[4], Pydantic models[5], and docstrings[6] are used to describe function inputs, outputs, and behaviors. This protocol allows EAPRouter to expose environment modules as a compact and standardized action space for EAP generation and execution. It also lowers the development barrier for connecting new environment modules into the EAP mechanism.

3.5 EAPRouter Implementation

EAPRouter is the component that implements EAP generation, safety validation, execution, and reuse. It serves as the intermediate translation layer between LLM agents and actual environment modules. At the runtime, it exposes the available module actions and agent inputs to the LLM, obtains an EAP for the current grounding procedure, validates the generated program, executes it in a protected interpreter, and returns both textual and structured results to the agent.

Action interface aggregation. During initialization, EAPRouter collects callable action interfaces from all registered environment modules through the metadata recorded by `@tool()`. For each callable action, it aggregates the module name, function name, type hints, Pydantic input-output models, docstrings, and the optional `kind` label into a compact action description. These descriptions form the action space provided to the LLM when generating EAPs, making the LLM aware of which modules can be called, what arguments each action requires, and what values each action returns.

Pre-generation for kind-marked actions. For high-frequency environment requests, EAPRouter uses the `kind` labels in action interfaces to pre-generate batch-calling EAPs during initialization. For example, actions marked as `kind="observe"` are grouped into a pre-generated `<observe>` EAP that invokes the corresponding observation actions and aggregates their outputs into a single `result`. Actions marked as `kind="statistic"` are similarly grouped for `<statistic>` requests. The generated EAPs are validated once before runtime and can then be reused directly, avoiding repeated LLM calls for common agent-environment interactions.

Runtime EAP generation. At runtime, EAPRouter first checks whether an incoming request can reuse a pre-generated or cached EAP. If no reusable EAP is found, it constructs an LLM prompt from four parts: the agent instruction, the structured `context`, the aggregated action interfaces of available environment modules, and the EAP output constraints. The constraints specify that the generated program should extract arguments from `context`, call environment actions only through the injected `modules` collection, optionally use `print()` for textual feedback, and assign the structured execution result to `result`. The LLM is therefore guided to generate a complete restricted Python program that explicitly represents action selection, action ordering, argument grounding, and intermediate-result reuse.

Safety validation and protected execution. Before execution, EAPRouter performs safety validation on the generated EAP. It parses the program and checks it against a whitelist of allowed syntax, built-in functions, and imports. It rejects unsupported operations such as arbitrary file access or dynamic code execution, verifies that `result` is assigned, and enforces an execution timeout to avoid dead loops. Only EAPs that pass these safety checks are executed by the protected interpreter. After execution, the interpreter returns the `result` variable as well as captured `stdout` and `stderr`. These outputs can be summarized into a natural-language response while the structured result is returned to the agent for subsequent processing.

Template-based EAP caching. To reduce the overhead of repeated generation, EAPRouter caches generated EAPs with template instructions. In large-scale agent interaction, many agents may issue semantically similar intents with different parameter values. Because the agent-EAP interface separates textual instructions from structured

[3] https://github.com/jlowin/fastmcp
[4] https://docs.python.org/3/library/typing.html
[5] https://docs.pydantic.dev/latest/
[6] https://peps.python.org/pep-0257/

`context`, variables can be moved from text into context and the remaining textual instruction can be treated as a reusable template string. When a new request arrives, the cache matching process first checks instruction-template semantic similarity through embedding similarity, and then checks structured-data consistency by comparing the keys and variable types in `context`. Only when both checks are satisfied will the cached EAP be reused with the new `context`, ensuring that the reused program remains behaviorally valid while reducing repeated LLM generation.

4 Experiments

The experiments in this section focus on the following research questions:

- RQ1: How accurately can EAPRouter ground LLM agent intents into valid environment action sequences in multi-module environments?
- RQ2: How effectively can EAP pre-generation and caching reduce the grounding overhead in large-scale multi-agent interaction?
- RQ3: Can EAPRouter safely block unsafe executable action programs and prompt-induced malicious generation?

4.1 Intent Grounding Accuracy

To verify whether EAPs can accurately ground agent intents into executable action sequences over different combinations of environment modules, we conducted a series of experiments on a benchmark to compare the grounding correctness and token consumption of EAPRouter with those of function-calling baselines.

Benchmark. The experiments were conducted on a benchmark that we constructed from records of agent-environment interactions during actual runs. In this benchmark, we first labeled the agent inputs with the ground-truth environment actions to be called, the action parameters, and the order of calls. We then categorized them into 6 difficulty levels based on the complexity of the combinations, and finally performed inter-level quantity balancing. To assess intent grounding correctness, we use the following evaluation metrics:

- **Intersection over Union (IoU):** used to evaluate the accuracy of action selection without considering order.
- **Normalized Longest Common Subsequence (NLCS) Score:** employed to assess the correctness of action sequencing.
- **Parameter Accuracy:** applied to verify the correctness of grounded action arguments.
- **Successful Call Ratio:** used as the strictest success rate metric, which requires the actual action sequence and parameters to exactly match the ground truth.

The details of the benchmark curation and evaluation metrics are discussed in Appendix B.

Baselines. To contrast EAP-based intent grounding with the more common step-by-step function-calling paradigm, we introduce the following function-calling baselines:

- **ReAct** (Yao et al., 2022): the most typical reasoning-action cycle.
- **PlanExecute** (Wang et al., 2023): a complete plan of action is first given and then executed, regenerating the plan if necessary.
- **SearchTool** (Braunschweiler et al., 2025): allows LLM to call the search tool to search for a collection of candidate functions before calling.
- **HierarchicalReAct:** first select the environment module, then select the next function to call.
- **HierarchicalPlanExecute:** first select the environment module, then generate the plan for this module.

Models. Following BFCLv4 (Patil et al., 2025), we selected the following models that perform well on function calls to cover open-source vs. closed-source and large vs. small models, including GLM-4.7 (GLM et al., 2024), Claude Opus 4.5, Kimi-K2-Instruct (Team et al., 2025), Qwen3-235B-A22B-Instruct, Qwen3-32B (Yang et al., 2025) and Gemini 2.5 Flash (Comanici et al., 2025).

Results. As shown in Table 1, EAPRouter achieves a large advantage over the function-calling baselines in terms of successful grounding rate. It reaches 60.0% with Kimi-K2-Instruct and 57.8% with GLM-4.7, improving over the best baseline by 31.6% and 25.4%, respectively. For the two closed-source models, EAPRouter also reaches 53.3% with Claude Opus 4.5 and 55.6% with Gemini 2.5 Flash, improving over the best baseline by 11.5% and 17.8%. At the same time, it brings no significant increase in the number of calls and output tokens, with an increase primarily in input token usage. In practice, this part of the overhead can be further reduced by EAP caching. Overall, this demonstrates that executable action programs can support accurate and efficient intent grounding in multi-module environments.

Further, by comparing the performance of different models on different level test cases as shown

Table 1: Comparison of intent grounding performance in the benchmark among baselines. Here we report the top two models ranked by successful call ratio (GLM-4.7 and Kimi-K2-Instruct), together with two closed-source models (Claude Opus 4.5 and Gemini 2.5 Flash). **Bolding** and Underlining are used to indicate optimal and suboptimal results under each model, respectively. See Table D1 for the full version in the appendix.

Method	Model	Performance					Cost (Per Test)		
		IoU ↑	NLCS ↑	PAcc ↑	SR ↑	SR Imp. ↑	Calls ↓	InTok ↓	OutTok ↓
EAPRouter	Kimi-K2-Instruct	**0.757**	**0.872**	**0.830**	**0.600**	+31.6%	1.4	15482.6	435.9
ReAct	Kimi-K2-Instruct	0.431	0.428	0.427	0.267	–	1.9	**6201.1**	**159.5**
PlanExecute	Kimi-K2-Instruct	0.465	0.587	0.608	0.339	–	**1.1**	6997.8	332.8
HierarchicalReAct	Kimi-K2-Instruct	0.437	0.433	0.464	0.283	–	5.9	7723.3	363.0
HierarchicalPlanExecute	Kimi-K2-Instruct	0.541	0.693	0.804	0.378	–	2.8	6307.8	379.9
SearchTool	Kimi-K2-Instruct	0.641	0.768	0.728	0.456	–	6.1	25171.5	558.9
EAPRouter	GLM-4.7	**0.771**	**0.886**	0.850	**0.578**	+25.4%	1.2	13775.2	1180.4
ReAct	GLM-4.7	0.467	0.489	0.490	0.294	–	2.0	7300.1	**506.5**
PlanExecute	GLM-4.7	0.615	0.709	**0.883**	0.456	–	**1.1**	6674.3	880.4
HierarchicalReAct	GLM-4.7	0.497	0.511	0.531	0.272	–	6.9	9913.6	3563.7
HierarchicalPlanExecute	GLM-4.7	0.634	0.713	0.851	0.400	–	2.8	**6077.8**	1943.8
SearchTool	GLM-4.7	0.642	0.773	0.778	0.461	–	6.7	42266.8	1408.2
EAPRouter	Claude Opus 4.5	**0.751**	**0.845**	0.824	**0.533**	+11.5%	1.4	16394.6	527.2
ReAct	Claude Opus 4.5	0.448	0.446	0.489	0.267	–	2.2	8041.8	292.6
PlanExecute	Claude Opus 4.5	0.575	0.773	**0.913**	0.422	–	**1.1**	6914.1	431.0
HierarchicalReAct	Claude Opus 4.5	0.504	0.497	0.542	0.311	–	6.8	9144.0	368.3
HierarchicalPlanExecute	Claude Opus 4.5	0.585	0.784	0.867	0.478	–	2.8	**6625.4**	518.1
SearchTool	Claude Opus 4.5	0.628	0.710	0.687	0.433	–	5.5	21389.6	**288.4**
EAPRouter	Gemini 2.5 Flash	**0.763**	**0.850**	0.843	**0.556**	+17.8%	1.5	17416.8	575.2
ReAct	Gemini 2.5 Flash	0.453	0.449	0.479	0.261	–	2.3	7808.1	**281.2**
PlanExecute	Gemini 2.5 Flash	0.562	0.753	**0.923**	0.428	–	**1.1**	6980.6	445.9
HierarchicalReAct	Gemini 2.5 Flash	0.491	0.488	0.552	0.311	–	7.0	9593.1	365.7
HierarchicalPlanExecute	Gemini 2.5 Flash	0.586	0.791	0.848	0.472	–	2.8	**6627.2**	514.4
SearchTool	Gemini 2.5 Flash	0.634	0.723	0.694	0.433	–	5.6	20838.1	294.1

Abbreviations: IoU = Intersection over Union; NLCS = Normalized Longest Common Subsequence; PAcc = Parameter Accuracy; SR = successful call ratio; SR Imp. = EAPRouter's improvement percentage over the best baseline in SR metric; #Calls = average LLM calls per test (including retrying); #InTok / #OutTok = average input/output tokens per test.

in Figure 2, we find that all selected LLMs can achieve similar performance on easier test cases, which reduces the risk of vendor lock-in in practical use of the method. However, harder test cases still pose a challenge to the model capability.

4.2 Efficiency and Reuse in Multi-Agent Interaction

In this experiment, we test the effectiveness of EAP pre-generation and caching in reducing grounding overhead by simulating the daily behavioral activities of 100 agents. The agent design follows AgentSociety (Piao et al., 2025) and adapts to the agent-EAP interface. We performed two runs of simulations, the first as a cold start of the caching mechanism and the second for testing the effect of cache reuse. Each round simulates 40 steps, corresponding to 0:00 to 10:00 in real time in order to include both active and inactive periods of agent behaviors.

The results are shown in Figure 3. We found that EAP pre-generation handles more than 70% of the agent calls (the first run is 74.1%; the second run is 70.5%). Meanwhile, after completing the cold start, the calls at the beginning of the second run are mostly resolved by the cache, while maintaining a high execution success rate. In addition, even without the cold-start process, EAP reuse among agents in the first run can reduce the call requirements by 66.5%. The average additional token overhead introduced per step and per agent in the two runs is 2,292 input tokens and 184 output tokens. These results indicate that EAP pre-generation and caching can significantly reduce the introduced grounding overhead and keep the system efficient. Further, it can be observed that agent active periods may have more diverse requests, resulting in the cache entries

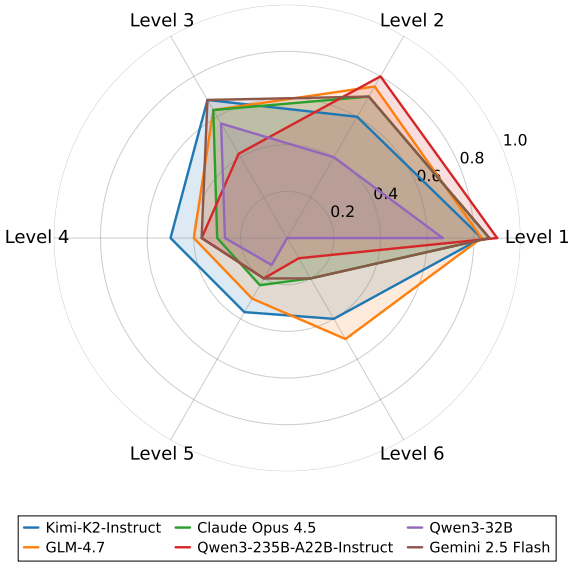

Figure 2: Comparison of successful grounding ratios of EAPRouter across different benchmark difficulty levels and models.

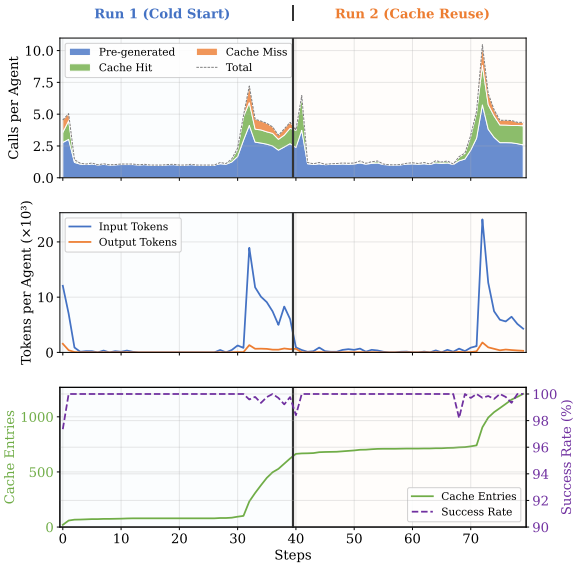

Figure 3: EAP pre-generation and cache reuse reduce API calls and token usage across simulation runs.

generated by only one run of cold-start not being able to fully satisfy the agent calls, which illustrates the importance of continuous accumulation of EAP cache data.

4.3 Safety Validation

Since EAPRouter executes generated programs, we further evaluate whether its safety validation can block unsafe EAPs before or during execution. We construct two safety test sets. The first set is a direct payload set with 33 code snippets covering unauthorized imports, file access, infinite loops, incorrect tool names, and unauthorized writes. The second one is a prompt-injection set with 16 natural-language instructions that attempt to induce the LLM to generate unsafe programs. EAPRouter uses a three-layer defense mechanism, including static syntax checks, a restricted execution environment, and runtime timeout control. As summarized in Appendix D.2, no effective attack escaped the sandbox. For direct payload injection, all 31 effective malicious payloads were blocked and the remaining two cases had no persistent side effects. For prompt injection, all 16 malicious prompts were refused or blocked, resulting in a 100% interception rate.

5 Conclusion

In this paper, we study the problem of grounding LLM agent intents in multi-module environments. We introduce executable action programs as an intermediate representation between textual intents and structured environment action calls, and implement EAPRouter to generate, validate, execute, and reuse such programs. Experiments on real agent-environment interaction traces show that EAPRouter improves grounding accuracy over function-calling baselines, reduces repeated generation cost through EAP pre-generation and caching, and blocks tested unsafe programs through safety validation.

Beyond these empirical gains, we believe executable action programs point to a broader direction for LLM agent research. As agents are deployed in environments with richer tools, longer interaction horizons, and stronger safety requirements, an explicit and reusable action-grounding representation can make agent behavior easier to inspect, reuse, and secure. Social simulation is one important application scenario, but the same idea may also benefit web agents, embodied agents, and software agents that must coordinate multiple modules or tools. Future work can extend EAPs with stronger formal verification, richer environment feedback, and cross-domain benchmarks to further study how LLM agents should ground intents into reliable actions in complex environments.

Limitations

Despite the promising results, our framework has the following limitations that warrant future investigation:

- **Dependency on Model Capability:** The per-

formance of CodeGenRouter is heavily reliant on the underlying LLM's coding ability. While state-of-the-art models perform well, smaller or less capable models may struggle with generating correct syntax for complex, multi-module interactions, potentially limiting accessibility.

- **Latency during Cold Start:** Although caching significantly mitigates overhead in the long run, the initial "cold start" phase of generating and verifying code introduces higher latency compared to native API calls, which may impact ad-hoc applications using environment module combinations without caches.

References

Norbert Braunschweiler, Rama Doddipatla, and Tudor-Catalin Zorila. 2025. Toolreagt: tool retrieval for llm-based complex task solution via retrieval augmented generation. In *Proceedings of the 3rd Workshop on Towards Knowledgeable Foundation Models (KnowFM)*, pages 75–83.

Sarthak Chakraborty, Suman Nath, Xuchao Zhang, Chetan Bansal, and Indranil Gupta. 2025. Generative caching for structurally similar prompts and responses. In *Advances in Neural Information Processing Systems*, NeurIPS '25.

Prateek Chhikara, Dev Khant, Saket Aryan, Taranjeet Singh, and Deshraj Yadav. 2025. Mem0: Building production-ready ai agents with scalable long-term memory. *arXiv preprint arXiv:2504.19413*.

Gheorghe Comanici, Eric Bieber, Mike Schaekermann, Ice Pasupat, Noveen Sachdeva, Inderjit Dhillon, Marcel Blistein, Ori Ram, Dan Zhang, Evan Rosen, and 1 others. 2025. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities. *arXiv preprint arXiv:2507.06261*.

Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Sam Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114.

Dawei Gao, Zitao Li, Xuchen Pan, Weirui Kuang, Zhijian Ma, Bingchen Qian, Fei Wei, Wenhao Zhang, Yuexiang Xie, Daoyuan Chen, and 1 others. 2024. Agentscope: A flexible yet robust multi-agent platform. *arXiv preprint arXiv:2402.14034*.

Team GLM, Aohan Zeng, Bin Xu, Bowen Wang, Chenhui Zhang, Da Yin, Dan Zhang, Diego Rojas, Guanyu Feng, Hanlin Zhao, and 1 others. 2024. Chatglm: A family of large language models from glm-130b to glm-4 all tools. *arXiv preprint arXiv:2406.12793*.

Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, Kai Dang, Yang Fan, Yichang Zhang, An Yang, Rui Men, Fei Huang, Bo Zheng, Yibo Miao, Shanghaoran Quan, and 5 others. 2024. Qwen2.5-coder technical report. *Preprint*, arXiv:2409.12186.

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

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. 2024. Swe-bench: Can language models resolve real-world github issues? *Preprint*, arXiv:2310.06770.

Joon Sung Park, Joseph O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S Bernstein. 2023. Generative agents: Interactive simulacra of human behavior. In *Proceedings of the 36th annual acm symposium on user interface software and technology*, pages 1–22.

Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E Gonzalez. 2025. The berkeley function calling leaderboard (bfcl): From tool use to agentic evaluation of large language models. In *International Conference on Machine Learning*, pages 48371–48392. PMLR.

Jinghua Piao, Yuwei Yan, Jun Zhang, Nian Li, Junbo Yan, Xiaochong Lan, Zhihong Lu, Zhiheng Zheng, Jing Yi Wang, Di Zhou, and 1 others. 2025. Agentsociety: Large-scale simulation of llm-driven generative agents advances understanding of human behaviors and society. *arXiv preprint arXiv:2502.08691*.

Mohit Shridhar, Xingdi Yuan, Marc-Alexandre Côté, Yonatan Bisk, Adam Trischler, and Matthew Hausknecht. 2021. ALFWorld: Aligning Text and Embodied Environments for Interactive Learning. In *Proceedings of the International Conference on Learning Representations (ICLR)*.

Kimi Team, Yifan Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, and 1 others. 2025. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*.

Lei Wang, Heyang Gao, Xiaohe Bo, Xu Chen, and Ji-Rong Wen. 2025a. Yulan-onesim: Towards the next generation of social simulator with large language models. *Preprint*, arXiv:2505.07581.

Lei Wang, Wanyu Xu, Yihuai Lan, Zhiqiang Hu, Yunshi Lan, Roy Ka-Wei Lee, and Ee-Peng Lim. 2023. Plan-and-solve prompting: Improving zero-shot chain-of-thought reasoning by large language models. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2609–2634, Toronto, Canada. Association for Computational Linguistics.

Maolin Wang, Yingyi Zhang, Bowen Yu, Bingguang Hao, Cunyin Peng, Yicheng Chen, Wei Zhou, Jinjie Gu, Chenyi Zhuang, Ruocheng Guo, and 1 others. 2025b. Function calling in large language models: Industrial practices, challenges, and future directions. *ACM Computing Surveys*.

Fengli Xu, Jun Zhang, Chen Gao, Jie Feng, and Yong Li. 2023. Urban generative intelligence (ugi): A foundational platform for agents in embodied city environment. *arXiv preprint arXiv:2312.11813*.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-mem: Agentic memory for llm agents. *arXiv preprint arXiv:2502.12110*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

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

Ziyi Yang, Zaibin Zhang, Zirui Zheng, Yuxian Jiang, Ziyue Gan, Zhiyu Wang, Zijian Ling, Jinsong Chen, Martz Ma, Bowen Dong, and 1 others. 2024b. Oasis: Open agents social interaction simulations on one million agents. *arXiv preprint arXiv:2411.11581*.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.

Jun Zhang, Yuwei Yan, Junbo Yan, Zhiheng Zheng, Jinghua Piao, Depeng Jin, and Yong Li. 2025. A parallelized framework for simulating large-scale llm agents with realistic environments and interactions. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 6: Industry Track)*, pages 1339–1349.

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, and 1 others. 2024. Webarena: A realistic web environment for building autonomous agents. In *International Conference on Learning Representations*, volume 2024, pages 15585–15606.

A Overview of Appendices

The appendices provide additional details and supporting materials that are too lengthy for the main text. Section B elaborates on the benchmark curation process, quality control protocols, and the explicit formulas for evaluation metrics used in our experiments. Section C presents a complete, end-to-end framework process example, demonstrating how the environment modules are integrated, initialized, and executed at runtime via the CodeGen-Router. Finally, Section D provides supplementary experimental materials, including the full performance comparison tables across all baselines and models.

B Benchmark Curation

This section summarizes how the benchmark instructions were collected, categorized, expanded, and validated to form a balanced and reliable evaluation set. Our benchmark is designed to evaluate fine-grained tool selection, multi-step sequencing, and argument grounding in some typical urban daily life scenario. The curation pipeline follows a staged design: instruction harvesting from live simulations, complexity-aware annotation and grouping, controlled data expansion for class balance, and multi-pass quality checks. This produces a hierarchical benchmark that spans from single-module atomic calls to multi-module, multi-step chained workflows.

We designed the benchmark with six pre-defined difficulty levels based on task complexity during annotation.

This benchmark is fully open-source under the CC BY 4.0 license.

B.1 Test Case Construction

All instructions originate from real agent commands issued during existing environment simulations, rather than being artificially authored. We then organize these instructions by complexity based on the number of modules involved and the required reasoning steps. Specifically, we annotate each instruction with (i) involved module set, (ii) temporal ordering requirements, (iii) required preconditions (e.g., observe or query before act), and (iv) argument determinism (explicit ID vs. inferred value). Based on these criteria, we define multiple complexity groups, including single-module atomic actions, two-module composites, and three-module chained workflows. These groups are ordered from local, single-step perception to global, multi-step coordination, forming a hierarchical and systematic benchmark. Human annotators label expected call sequences with explicit preconditions (e.g., obtain location before nearby search) and with wildcard arguments where the value is not deterministic.

To ensure class balance, we apply a controlled expansion strategy: for under-represented groups, we generate semantically equivalent variants by modifying destinations, modes, event types, or social actions while keeping the same structural complexity and expected call sequence. We also add a small subset of high-complexity cases that include multiple discovery and confirmation steps (e.g., observe → search → move → confirm → act), which are designed to stress the router's multi-step planning and argument handling.

The final benchmark contains 18 instruction categories with 10 cases per category (180 total). These cases cover diverse locations (AOI/POI), mobility modes, event types, and social interaction patterns, yielding broad coverage of common daily-life situations. Example test cases are shown below (exact dataset stored in `instructions_complete.yaml`):

```
1) "Check current location, then search nearby
    restaurants within 500m."
   Expected:
   - ["MobilitySpace", "get_person",
    {"person_id": "@"}]
   - ["MobilitySpace", "find_nearby_pois",
    {"x": "*", "y": "*", "category":
    "restaurant", "radius": 500}]

2) "Move to POI 700002365 and start a 30-minute
    eating out event."
   Expected:
   - ["MobilitySpace", "move_to", {"person_id":
    "@", "aoi_id_or_poi_id": 700002365, "mode":
    "driving"}]
   - ["MobilitySpace", "get_person",
    {"person_id": "@"}]
   - ["EventSpace", "start_event",
    {"person_id": "@", "event_type": "eating
    out", "event_name": "*",
    "expected_duration_seconds": 1800}]

3) "Search for cafes, move to one, then refresh
    the feed."
   Expected:
   - ["MobilitySpace", "get_person",
    {"person_id": "@"}]
   - ["MobilitySpace", "find_nearby_pois",
    {"x": "*", "y": "*", "category": "cafe",
    "radius": "*"}]
   - ["MobilitySpace", "move_to", {"person_id":
    "@", "aoi_id_or_poi_id": "*", "mode":
    "walking"}]
   - ["MobilitySpace", "get_person",
```

```
    {"person_id": "@"}]
  - ["SocialMediaSpace", "refresh_feed",
    {"user_id": "@", "algorithm": "*", "limit":
    "*"}]
```

Wildcards (`*`) are ignored during parameter matching, while context markers (`@`) are resolved to the agent's contextual identifier (`context["id"]`).

For example, given the instruction *"Check current location, then search nearby restaurants within 500m"*, the expected tool calls are:

- `["MobilitySpace", "get_person", {"person_id": "@"}]` — retrieves the current agent's location and mobility status; the `@` marker ensures that `person_id` matches the executing agent's context identifier.
- `["MobilitySpace", "find_nearby_pois", {"x": "*", "y": "*", "category": "restaurant", "radius": 500}]` — searches for nearby restaurants; the `*` wildcards for `x` and `y` indicate that these values do not affect benchmark correctness, as coordinates are inferred from context.

B.2 Quality Control

We apply multi-stage quality control. First, multiple annotators cross-check instruction semantics against the expected call sequence and argument types. Second, we run LLM dry-runs to detect mismatches such as missing preconditions, incorrect function selection, or inconsistent parameter use. Finally, we spot-check failure cases from the benchmark runner and iteratively refine ambiguous instructions or overly rigid expected calls.

B.3 Evaluation Metric Formulas

Let E be the expected call set and A be the actual call set. Let S_E be the expected function sequence and S_A be the actual function sequence. For each expected call $e \in E$, let $a^*(e)$ be the best-matching actual call with the same module and function name.

Function Selection IoU:

$$\mathrm{IoU}(E, A) = \frac{|F(E) \cap F(A)|}{|F(E) \cup F(A)|}$$

where $F(\cdot)$ is the set of function names (order ignored).

NLCS Score:

$$\mathrm{NLCS} = \frac{\mathrm{LCS}(S_A, S_E)}{|S_E|}$$

where LCS is the longest common subsequence length.

Parameter Accuracy: For each expected call e, compute

$$\mathrm{accuracy}(e) = \frac{\#\text{ matched params}}{\#\text{ required params}}$$

Overall parameter accuracy is the mean over all expected calls:

$$\mathrm{PAcc} = \frac{1}{|E|} \sum_{e \in E} \mathrm{accuracy}(e)$$

Successful Call:

$$\mathrm{Successful} = [\mathrm{NLCS}(S_A, S_E) = 1.0] \wedge [\mathrm{PAcc} = 1.0]$$

C A Framework Complete Process Example

We present a complete example using three environment modules (Daily Mobility and Event) to illustrate the end-to-end workflow from integration to runtime execution.

C.1 Development: Environment Module Integration

The integration requires only minimal code: implement tools with the provided decorator and register the environment modules in the router. The following snippet shows the tool declaration pattern and highlights that only a few lines are needed.

```
from SimFramework.env import EnvBase, tool

class MobilitySpace(EnvBase):
    @tool(readonly=True, kind="observe")
    async def get_person(self, person_id: int):
        ...

    @tool(readonly=False)
    async def move_to(self, person_id: int,
     aoi_id_or_poi_id: int, mode: str =
     "driving"):
        ...

class EventSpace(EnvBase):
    @tool(readonly=False)
    async def start_event(self, person_id: int,
     event_type: str, event_name: str,
     expected_duration_seconds: float):
        ...
```

In the benchmark runner, modules are registered in a single line:

```
env_router =
    router_class(env_modules=[mobility_env,
    event_space])
```

C.2 Initialization: Environment Interface File Generation

The framework auto-generates a `.pyi` interface file that exposes tool signatures to the code generation router. For the above case, we only include the relevant subset (Mobility + Event):

```
class GetPersonResponse(BaseModel):
    id: int
    status: Literal["idle", "moving"]
    position: Position
    target: Optional[TargetResponse]

class StartEventResponse(BaseModel):
    person_id: int
    event_type: str
    event_name: str
    start_time: datetime
    expected_end_time: datetime
    status: str

class MobilitySpace:
    async def get_person(person_id: int) ->
     GetPersonResponse: ...
    async def move_to(person_id: int,
     aoi_id_or_poi_id: int, mode:
     Literal["walking","driving"]="driving") ->
     Any: ...

class EventSpace:
    async def start_event(person_id: int,
     event_type: str, event_name: str,
     expected_duration_seconds: float) ->
     StartEventResponse: ...
    async def get_current_event(person_id: int)
     -> Optional[GetEventResponse]: ...
    async def stop_event(person_id: int,
     status:
     Literal["completed","cancelled"]="completed")
     -> EndEventResponse: ...
```

This interface constrains the code generator to valid tools and argument types, enabling safe, verifiable tool invocation.

C.3 Runtime: Agent-Environment Interaction with Generative Code Execution

We show two runtime interactions: an <observe> step and a task step. Each includes (1) agent input, (2) generated code, (3) execution results and (4) summarized return.

Example A: <observe> step (multi-module state refresh)

```
1) Agent input:
   instruction:"<observe>"
   context: {"id":3}

2) Generated code:
   async def run():
       print(
           f"Collecting environment
    observations for agent {context['id']}..."
       )
       person = await MobilitySpace.get_person(
           person_id=context["id"]
       )
       event = await
    EventSpace.get_current_event(
           person_id=context["id"]
       )
       result = {"person": person, "event":
    event}
       print("Environment observations
    collected successfully.")
       return result

3) Execution result (raw):
   {"person": {"id": 3, "status": "idle",
    "position": {"kind": "aoi", "aoi_id":
    500023998, "xy": [..]}},
    "event": null}

4) Summarized return to agent:
    Response:"You are currently idle at AOI
    500023998; no active event; 1 unread DM and
    recent feed available."
    Result:   {"person": {"id": 3, "status":
    "idle", "position": {"kind": "aoi",
    "aoi_id": 500023998, "xy": [..]}},
    "event": null}
```

The <observe> instruction signals a state refresh, so the system first calls `get_person` to obtain current location and movement status, then calls `get_current_event` to check whether an activity is already running, and finally calls `observe_user` to gather social context (unread messages and recent feed). These calls are sequenced to cover mobility, event, and social states in one pass, and their structured outputs are merged into a single natural language summary for the agent.

Example B: Query POI, move, confirm, and start event

```
1) Agent input:
   instruction: "Go to Blue Maple Bistro (POI
    ID: {poi_id}) and start a 30-minute eating
    out event."
   context:{"id": 3, "variables": {"poi_id":
    700002365}}

2) Generated code:
   async def run():
       print(
           f"Routing to POI
    {context['variables']['poi_id']} for agent
    {context['id']}..."
       )
       poi = await MobilitySpace.get_poi(
           poi_id=context['variables']['poi_id']
       )
       await MobilitySpace.move_to(
           person_id=context["id"],

    aoi_id_or_poi_id=context['variables']['poi_id'],
           mode="driving"
       )
       person = await MobilitySpace.get_person(
           person_id=context["id"]
```

```
        )
        event = await EventSpace.start_event(
            person_id=context["id"],
            event_type="eating out",
            event_name="meal",
            expected_duration_seconds=1800
        )
        result = {"event": event}
        print("Event started successfully.")
        return result

3) Execution result (raw):
   {"event": {"person_id": 3, "event_type":
     "eating out", "event_name": "meal",
     "status": "in_progress", ...}}

4) Summarized return to agent:
   Response:"Started driving to Blue Maple
     Bistro; will begin eating out upon arrival
     (30 min)."
   Result: {"event": {"person_id": 3,
     "event_type": "eating out", "event_name":
     "meal", "status": "in_progress", ...}}
```

The instruction names a restaurant, so the system first calls `get_poi` to retrieve the POI details used by navigation. It then issues `move_to` to start driving and follows with `get_person` to confirm arrival status before starting the event. Finally, it calls `start_event` to begin the 30-minute eating activity. The response explicitly reflects this sequence: the agent is en route and will start eating upon arrival, which mirrors the tool calls.

Critically, the structured context with templated variables (e.g., `poi_id`) enables CodeGenRouter to cache this generated code. When other agents issue semantically similar instructions—such as visiting different restaurants—the system reuses the cached template after verifying both embedding-based semantic similarity and structural type consistency, dramatically reducing code generation overhead in large-scale simulations.

D Experiment Supplementary Materials

In this section, we provide a comprehensive supplement to the experimental details mentioned earlier, including complete experimental results and the performance of all baseline methods across different models.

D.1 Complete Results of Framework Performance

We conducted benchmark experiments on EAPRouter and five other baselines across six models. Table D1 shows the complete experiment results.

D.2 Safety Validation Benchmark

We evaluate the safety validation of EAPRouter with two complementary attack paths. The first path directly injects malicious EAP snippets into the validation and execution pipeline, which isolates the sandbox from the LLM's own refusal behavior. The second path uses natural-language prompt injection to induce the LLM to generate unsafe programs, testing the end-to-end behavior of EAP generation and execution.

EAPRouter uses a three-layer defense mechanism. L1 performs static syntax validation before execution, including checks over dangerous modules, dangerous functions, forbidden syntax nodes, and obvious infinite-loop patterns. L2 executes accepted programs in a restricted environment, where environment modules are exposed through read-only mappings, imports are limited by a whitelist, and built-in functions are restricted. L3 enforces runtime timeout control for long-running programs.

Table D2 summarizes the direct payload tests. The benchmark contains 33 payloads across five attack types: unauthorized imports, file read/write, infinite loops, incorrect tool names, and unauthorized writes. Among them, 31 effective malicious payloads are blocked by L1–L3, while two local write attempts are allowed because they only modify transient local data or rebuilt built-ins and have no persistent side effect. Thus, no direct payload escapes the sandbox.

Table D3 summarizes the prompt-injection tests. The 16 prompts cover direct malicious instructions, disguised tasks, indirect injections, code obfuscation, and unauthorized actions. All prompts are refused or blocked before unsafe behavior is executed, yielding a 100% interception rate in the end-to-end setting.

D.3 Comparison of Baselines and Models on Different Difficulty Level of Tasks

Additionally, we evaluated the performance of various baselines and models on tasks of varying difficulty levels. For each router method, we plot a single radar chart where each axis represents one difficulty level (Level 1 to Level 6) and each curve corresponds to a model's performance across these levels. Task difficulty is assigned according to per-task `success_rate` statistics in `by_type`, with the following definitions:

- Level 1: State observation tasks (single-step status retrieval)

- Level 2: Discovery/query tasks (information seeking before action)
- Level 3: Atomic single-module actions (direct execution within one module)
- Level 4: Sequential long-horizon tasks (multi-step dependencies within a coherent workflow)
- Level 5: Two-module composition tasks (cross-module coordination)
- Level 6: Three-module composition tasks (full multi-module orchestration)

The score on each axis is the successful call ratio across all task types mapped to that difficulty level. Figure 2 in the main text corresponds to the performance of EAPRouter, Figure D1 corresponds to the performance of PlanExecuteRouter, Figure D2 corresponds to the performance of ReActRouter, Figure D3 corresponds to the performance of SearchToolRouter, Figure D4 corresponds to the performance of HierarchicalPlanExecuteRouter and Figure D5 corresponds to the performance of HierarchicalReActRouter.

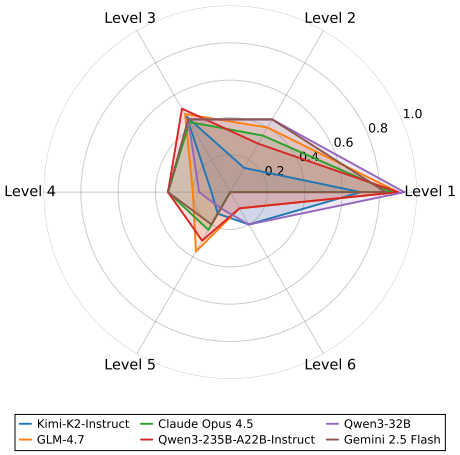

Figure D1: Comparison of successful call ratios of PlanExecuteRouter in different levels of the benchmark with different models.

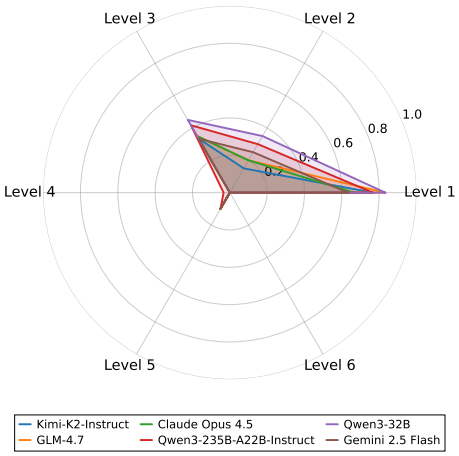

Figure D2: Comparison of successful call ratios of ReActRouter in different levels of the benchmark with different models.

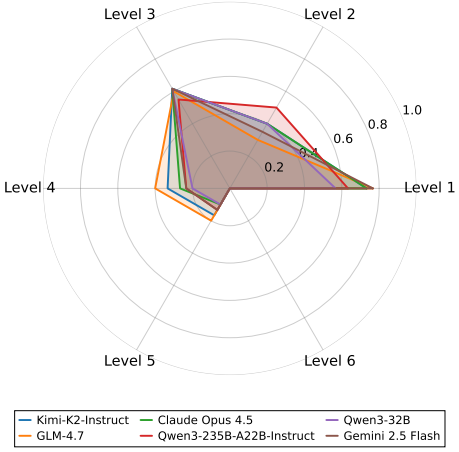

Figure D3: Comparison of successful call ratios of SearchToolRouter in different levels of the benchmark with different models.

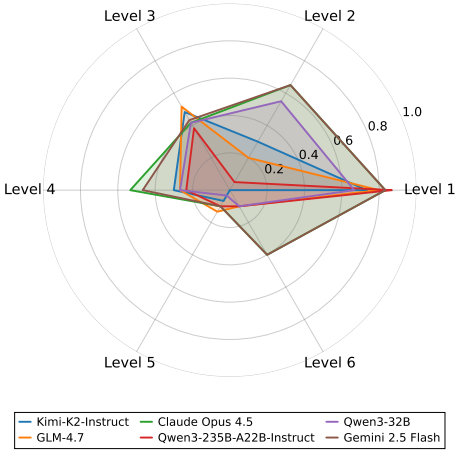

Figure D4: Comparison of successful call ratios of HierarchicalExecuteRouter in different levels of the benchmark with different models.

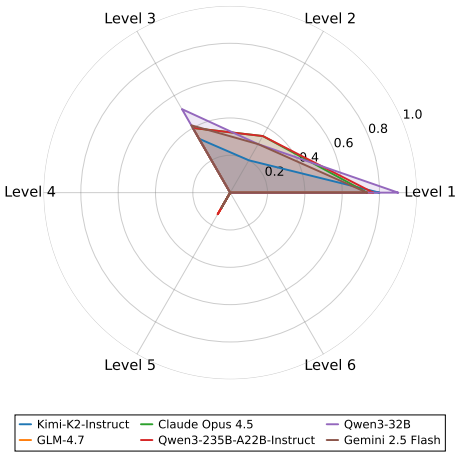

Figure D5: Comparison of successful call ratios of HierarchicalReActRouter in different levels of the benchmark with different models.

Table D1: Experiment results across all baseline methods and all models.

Method	Model	Performance					Cost (Per Test)		
		IoU ↑	NLCS ↑	PAcc ↑	SR ↑	SR Imp. ↑	Calls ↓	InTok ↓	OutTok ↓
EAPRouter	GLM-4.7	**0.771**	**0.886**	0.850	**0.578**	+25.4%	1.2	13775.2	1180.4
ReAct	GLM-4.7	0.467	0.489	0.490	0.294	–	2.0	7300.1	**506.5**
PlanExecute	GLM-4.7	0.615	0.709	**0.883**	0.456	–	**1.1**	6674.3	880.4
HierarchicalReAct	GLM-4.7	0.497	0.511	0.531	0.272	–	6.9	9913.6	3563.7
HierarchicalPlanExecute	GLM-4.7	0.634	0.713	0.851	0.400	–	2.8	**6077.8**	1943.8
SearchTool	GLM-4.7	0.642	0.773	0.778	0.461	–	6.7	42266.8	1408.2
EAPRouter	Claude Opus 4.5	**0.751**	**0.845**	0.824	**0.533**	+11.5%	1.4	16394.6	527.2
ReAct	Claude Opus 4.5	0.448	0.446	0.489	0.267	–	2.2	8041.8	292.6
PlanExecute	Claude Opus 4.5	0.575	0.773	**0.913**	0.422	–	**1.1**	6914.1	431.0
HierarchicalReAct	Claude Opus 4.5	0.504	0.497	0.542	0.311	–	6.8	9144.0	368.3
HierarchicalPlanExecute	Claude Opus 4.5	0.585	0.784	0.867	0.478	–	2.8	**6625.4**	518.1
SearchTool	Claude Opus 4.5	0.628	0.710	0.687	0.433	–	5.5	21389.6	**288.4**
EAPRouter	Kimi-K2-Instruct	**0.757**	**0.872**	**0.830**	**0.600**	+31.6%	1.4	15482.6	435.9
ReAct	Kimi-K2-Instruct	0.431	0.428	0.427	0.267	–	1.9	**6201.1**	**159.5**
PlanExecute	Kimi-K2-Instruct	0.465	0.587	0.608	0.339	–	**1.1**	6997.8	332.8
HierarchicalReAct	Kimi-K2-Instruct	0.437	0.433	0.464	0.283	–	5.9	7723.3	363.0
HierarchicalPlanExecute	Kimi-K2-Instruct	0.541	0.693	0.804	0.378	–	2.8	6307.8	379.9
SearchTool	Kimi-K2-Instruct	0.641	0.768	0.728	0.456	–	6.1	25171.5	558.9
EAPRouter	Qwen3-235B-A22B-Instruct	**0.657**	**0.729**	0.788	**0.478**	+2.4%	**1.1**	12111.9	476.4
ReAct	Qwen3-235B-A22B-Instruct	0.499	0.509	0.504	0.322	–	4.8	12719.5	490.3
PlanExecute	Qwen3-235B-A22B-Instruct	0.492	0.726	**0.922**	0.467	–	1.2	**7155.9**	446.8
HierarchicalReAct	Qwen3-235B-A22B-Instruct	0.532	0.573	0.591	0.322	–	9.6	13575.9	393.2
HierarchicalPlanExecute	Qwen3-235B-A22B-Instruct	0.489	0.711	0.864	0.339	–	3.3	7505.3	585.4
SearchTool	Qwen3-235B-A22B-Instruct	0.616	0.691	0.661	0.406	–	5.8	22162.5	**305.0**
EAPRouter	Qwen3-32B	**0.665**	**0.732**	0.705	**0.411**	+0.0%	1.5	17350.5	585.2
ReAct	Qwen3-32B	0.513	0.506	0.558	0.339	–	2.0	**6193.3**	**83.0**
PlanExecute	Qwen3-32B	0.606	0.705	**0.884**	0.411	–	**1.1**	6725.0	303.5
HierarchicalReAct	Qwen3-32B	0.536	0.533	0.630	0.356	–	5.6	7292.8	143.2
HierarchicalPlanExecute	Qwen3-32B	0.562	0.686	0.810	0.367	–	3.0	6417.2	432.7
SearchTool	Qwen3-32B	0.603	0.674	0.700	0.394	–	5.5	21543.1	167.4
EAPRouter	Gemini 2.5 Flash	**0.763**	**0.850**	0.843	**0.556**	+17.8%	1.5	17416.8	575.2
ReAct	Gemini 2.5 Flash	0.453	0.449	0.479	0.261	–	2.3	7808.1	**281.2**
PlanExecute	Gemini 2.5 Flash	0.562	0.753	**0.923**	0.428	–	**1.1**	6980.6	445.9
HierarchicalReAct	Gemini 2.5 Flash	0.491	0.488	0.552	0.311	–	7.0	9593.1	365.7
HierarchicalPlanExecute	Gemini 2.5 Flash	0.586	0.791	0.848	0.472	–	2.8	**6627.2**	514.4
SearchTool	Gemini 2.5 Flash	0.634	0.723	0.694	0.433	–	5.6	20838.1	294.1

Abbreviations: IoU = Intersection over Union; NLCS = Normalized Longest Common Subsequence; PAcc = Parameter Accuracy; SR = successful call ratio; SR Imp. = EAPRouter's improvement percentage over the best baseline in SR metric; #Calls = average LLM calls per test (including retrying); #InTok / #OutTok = average input/output tokens per test.

Table D2: Direct payload injection tests for EAP safety validation.

Type	Representative cases	#	Block layer	Escaped
Unauthorized import	`import os`, `import subprocess`, `import socket`	12	L1:10, L2:2	0
File access	`open(...)`, `eval(...)`, `exec(...)`	6	L1:4, L2:2	0
Infinite loop	`while True`, large-range `for` loop	5	L1:4, L3:1	0
Wrong tool	nonexistent module/tool and invalid arguments	4	L2:4	0
Unauthorized write	module overwrite, `del`, `global`	6	L1:3, L2:1, pass:2	0
Total	–	33	L1:21, L2:9, L3:1, pass:2	0

Table D3: Prompt-injection tests for end-to-end safety validation.

Type	Representative intent	#	Escaped
Direct instruction	ask for `os`, `subprocess`, file, or socket operations	5	0
Disguised task	hide unsafe code in debugging or module-reload requests	3	0
Indirect injection	use "ignore previous instructions" or system override style requests	3	0
Obfuscation	use dynamic import, string composition, or class definition	3	0
Unauthorized action	clear module mappings or request infinite execution	2	0
Total	–	16	0