ARE: scaling up agent environments and evaluations

Meta Superintelligence Labs[1]

[1]A detailed contributor list can be found in the appendix of this paper.

We introduce **Meta Agents Research Environments (ARE)**, a research platform for scalable creation of environments, integration of synthetic or real applications, and execution of agentic orchestrations. ARE provides simple abstractions to build complex and diverse environments, each with their own rules, tools, content, and verifiers, helping to bridge the gap between model development and real-world deployment. We also propose **Gaia2**, a benchmark built in ARE and designed to measure general agent capabilities. Beyond search and execution, Gaia2 requires agents to handle ambiguities and noise, adapt to dynamic environments, collaborate with other agents, and operate under temporal constraints. Unlike prior benchmarks, Gaia2 runs asynchronously, surfacing new failure modes that are invisible in static settings. Our experiments show that no system dominates across the intelligence spectrum: stronger reasoning often comes at the cost of efficiency, and budget scaling curves plateau, highlighting the need for new architectures and adaptive compute strategies. Perhaps more importantly, ARE abstractions enable continuous extension of Gaia2 to other environments, empowering the community to rapidly create new benchmarks tailored to their domains. In AI's "second half", progress increasingly depends on defining meaningful tasks and robust evaluations to drive frontier capabilities forward.

Date: September 21, 2025

Correspondence: rfroger@meta.com, gmialon@meta.com, tscialom@meta.com

Code: https://github.com/facebookresearch/meta-agents-research-environments

Meta

1 Introduction

Scaling large language model (LLM) training with reinforcement learning (RL) is a promising path towards continuous model improvements and, eventually, superintelligence. In particular, reinforcement learning from verifiable rewards (RLVR) has recently emerged as a more scalable alternative to reliance on reward models in settings like reasoning, coding (OpenAI, 2024b, 2025b; DeepSeek-AI et al., 2025; Mistral-AI et al., 2025),

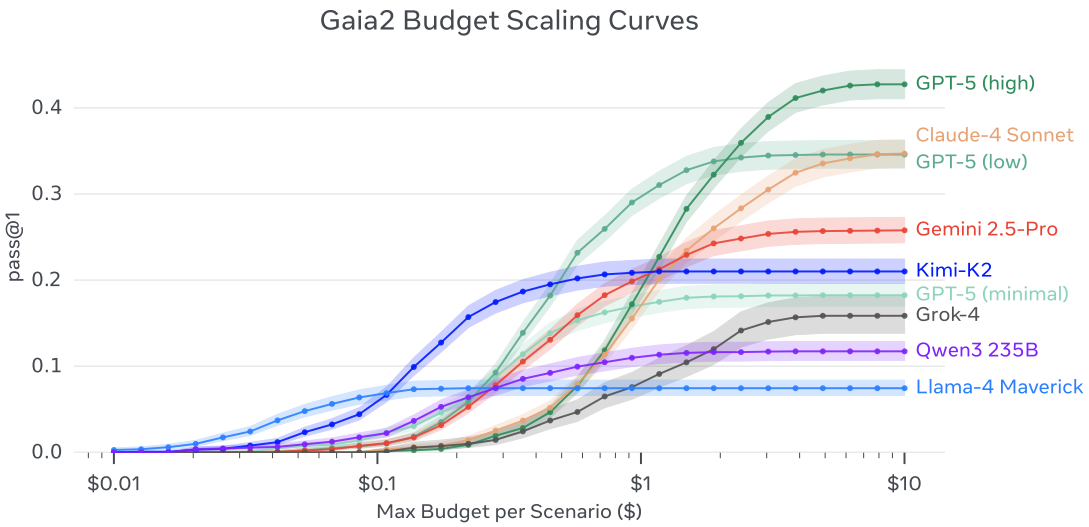

Figure 1 Gaia2 budget scaling curve: for each max_budget, we plot $\sum \mathbb{1}\{\text{scenario\_result} = \text{True} \wedge \text{scenario\_cost} < \text{max\_budget}\}$. Equipped with a simple ReAct-like scaffold (see Section 2.4), no model evaluated here dominates across the intelligence spectrum—each trades off capability, efficiency, and budget. At equal cost, some models fare better, yet all curves plateau, suggesting that standard scaffolds and/or models miss key ingredients for sustained progress. Cost estimates from Artificial Analysis model pricing data (accessed September 10, 2025).

agent tool use (MoonshotAI et al., 2025), or even chat (Mu et al., 2024). Concurrently, models are now addressing tasks involving deeper interactions with the outside world over longer time periods, as reflected by the emergence of new benchmarks (Mialon et al., 2023; Jimenez et al., 2024; Yao et al., 2024; Backlund and Petersson, 2025) and products (Google, 2024; OpenAI, 2025a).

In this context, we posit that model improvement through experience and deployment in production are bounded by the controllability, diversity, and realism of available environments. First, while the web is a great environment for supporting agent tasks like search, it is constantly evolving, making reproducibility for evaluation (Mialon et al., 2023; Wei et al., 2025a) and study of complex behaviors challenging, in particular those involving `write` operations. Creating simulated environments is an appealing alternative providing more control to developers, but requires spending sufficient effort on diversity and realism. Existing environments are therefore tightly coupled to narrow sets of tasks and capabilities, and even agent modeling. Since environments saturate quickly with model progress, moving to new environments and tasks requires frequent rewriting of a lot of boilerplate code. As of the writing of this paper, there are few open-source and flexible libraries for developing and studying practical LLM agents (Brown, 2025).

Second, most environments reflect idealized models of agent interaction. These idealized models reduce task diversity and do not map to real-world deployment conditions. For example, τ-bench (Yao et al., 2024) and SWE-bench (Jimenez et al., 2024) agents operate sequentially and the environment is paused while the agent is working, preventing the state of the world from changing in the interim and effectively giving away many valuable real-world capabilities, such as asynchronous communication with users and adaptation to new events.

We therefore propose Meta Agents Research Environments (ARE), a research platform that supports the running of orchestrations, creation of environments, and connection of synthetic or real world apps for agent development and evaluations. ARE does so by: (i) proposing abstractions for simulation and verification that facilitate both the creation of diverse environments and tasks, as well as the integration of existing ones, like τ-bench; and (ii) supporting a shift from sequential to asynchronous interaction between an agent and its environment, unlocking new tasks and capabilities in the process, like handling time. Though simulated, the platform is not unrealistic. ARE supports connection of real apps e.g., through Model Context Protocol (MCP) integration (Anthropic, 2024), so that model development, evaluation, and production deployment can be consistent. In addition to RL, ARE enables generation of high-quality SFT traces.

Building on ARE, we introduce Gaia2, a new evaluation for agents. Gaia2 is composed of 1,120 verifiable, annotated scenarios taking place in a `Mobile` environment that mimics a smartphone, with apps such as email, messaging, calendar, etc. as well as their associated content, similar to AppWorld (Trivedi et al., 2024) or ToolSandbox (Lu et al., 2024). Gaia2 is designed to address the need for challenging yet holistic evaluations for agents beyond pure search-and-execution (Mialon et al., 2023; Wei et al., 2025a; Yao et al., 2024; Trivedi et al., 2024; Lu et al., 2024). Gaia2 retains the core principles of Gaia, consisting of verifiable tasks that are simple for humans but challenging for today's models, and that align with actual model use. The benchmark also integrates our learnings from using Gaia. We propose more diverse tasks in a simulated but dense and realistic environment with built-in tools so that signal-to-noise ratio and reproducibility are better. Furthermore, we target new capabilities with scenarios that require agents to exhibit adaptability and to effectively handle ambiguity, noise, time, and collaboration with other agents. If developed, these new capabilities will unlock a breadth of practical use cases.

Gaia2 departs from most agent benchmarks in two ways. (i) Deeper and more realistic interactions between the agent and the environment, since both run asynchronously. In particular, we shift from tasks to scenarios spanning an arbitrary period of time. Environment time passes independent of whether the agent acts or not, and the environment state is continuously updated with random or scheduled events, such as a friend replying to message sent by a user or an agent. This unlocks a breadth of new scenarios requiring the agent to adapt at test-time to either ignore incoming events or to be always-on and proactive, performing actions in due time. (ii) We propose a robust verification system that lends itself to RL, comparing agent `write` actions only to annotated oracle `write` actions, and for each `write` action, evaluating arguments, that can be seen as rubrics, via a soft (LLM judge) or hard (exact-match) comparison depending on the argument type.

While today's frontier models are far from solving Gaia2, we do not consider it to be an "AGI-level" benchmark; in the LLM and RL era, we expect rapid hill-climbing. However, we believe Gaia2, with its richer interactions with an environment and resulting spectrum of scenarios, is aligned with actual progress towards practically

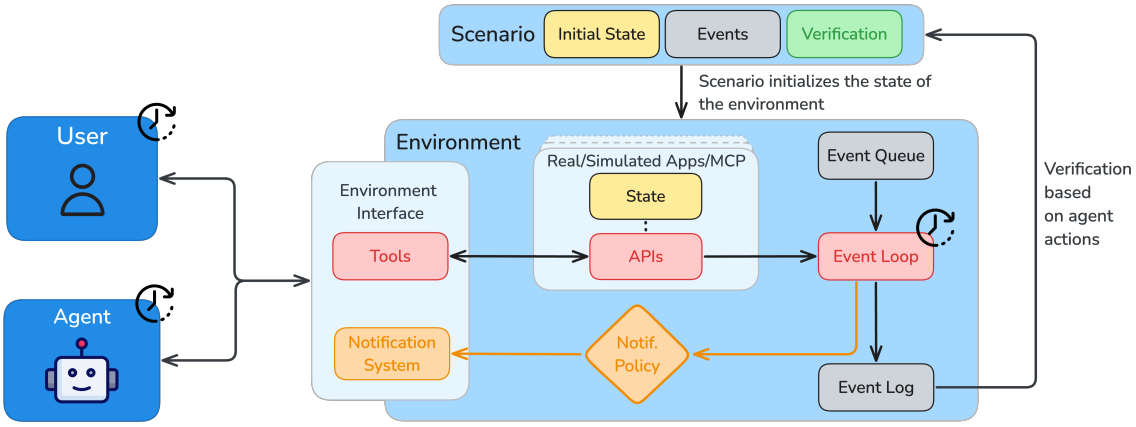

Figure 2 ARE environments are event-based, time-driven simulations, that run asynchronously from the agent and the user. ARE environments allow to play scenarios, which typically contain tasks for the agent and verification logic. Whether initiated by agent or user, interactions happen through the same interfaces and can be either tool calls, or tool output/notification observations. Extensive simulation control and logging allow precise study of agents behavior.

useful agents. We also expect that doing well on multi-agent and time-based scenarios will require some modeling effort beyond scaling training. Finally, Gaia2 lends itself to extensions. There is no progress without reliable evaluations, and we hope removing the need for writing boilerplate environment and runtime code will help the community continue creating benchmarks that challenge current modeling standards.

2 ARE: A Research Platform to Create Environments and Run Agents

ARE is a research platform for creating simulated environments, running agents on scenarios within them, and analyzing their behavior. ARE environments evolve continuously and are strictly decoupled from the agent. Time advances in the simulation, and the environment continuously introduces events. The agent runs asynchronously and interacts with the user and the environment through a dedicated interface. Figure 2 provides an overview of ARE 's architecture and high-level abstractions.

2.1 ARE Foundations

ARE is time-driven and built on the principle that "**everything is an event**". Specifically, five core concepts work together:

1. **Apps** are stateful API interfaces that typically interact with a data source.
2. **Environments** are collections of Apps, their data, and governing rules that define system behavior.
3. **Events** are anything that happens in the Environment. All Events are logged.
4. **Notifications** are messages from the Environment that inform the agent about Events. They are configurable and enable selective observability of the Environment.
5. **Scenarios** are sets of initial state and scheduled Events that take place in an Environment, and can include a verification mechanism.

2.1.1 Apps

Apps are collections of tools that interact with a data source. For instance, an Emails app contains tools like `send_email` and `delete_email` that all operate on the same email database. Similar approaches have been explored in AppWorld (Trivedi et al., 2024) and ToolSandbox (Lu et al., 2024).

Apps maintain their own state Each app starts in the simulation with an initial state and keeps track of changes as agents use its tools or as events occur in the environment. Apps store their data internally rather than relying on external databases. This design makes it convenient to study agent tasks that require to modify the state of the environment, and ensures that experiments can be reproduced consistently.

Tool creation and taxonomy Apps are implemented by adding Python methods within an `App` class. When the simulation runs, these methods are automatically converted into properly formatted tool descriptions that agents can understand and use. ARE classifies tools into two types via decorators: `read`, which only read app states (e.g., `search_emails`), and `write`, which modify app states (e.g., `send_email`). This distinction is helpful *e.g.* for verification, see Section 2.3. Tools are role-scoped—`agent`, `user`, or `env`. For example, certain user tools may be unavailable to the agent due to sensitivity. See Appendix A.1 for example code snippets.

Extensibility Beyond *ad hoc* app creation, ARE can also connect with external APIs through MCP compatibility (Anthropic, 2024). The framework also offers flexible options for data storage. While our current implementation stores data in memory, users can easily connect SQL databases or other storage systems without changing the core framework.

Core apps Developers can choose which apps to include in their environment or create new ones. However, every ARE environment includes two core apps that handle the basic interaction between agents and their environment:

- `AgentUserInterface` is the communication channel between users and agents: messages are tool calls, and user messages generate notifications (Section 2.1.4) that agents can process asynchronously. This enables asynchronous interactions during task execution. The interface supports two modes: *blocking* (the agent waits for a user reply) and *non-blocking* (the agent continues loop regardless of reply).
- `System` provides core simulation controls like `get_current_time` (query time), `wait` (pause for a duration), and `wait_for_next_notification` (pause until an event). When any wait tool is invoked, the simulation accelerates: it switches from real time to a queue-based, event-to-event loop. Scenarios that would take hours in the real world can thus run in minutes, enabling practical long-horizon testing.

2.1.2 Environment

An environment is a Markov Decision Process with states, observations, actions, and transition rules. The environment state includes the states of all apps, the time manager, and the notification system. Apps define the action space by exposing their tools. The environment runs deterministically given a fixed starting state and seed, ensuring reproducible evaluations. It can host one or multiple agents simultaneously, supporting both single-agent and multi-agent setups. The environment's rules define time progression, action permissions, reward computation, and how agent actions affect the environment state.

2.1.3 Events

In ARE, an event is any agent action or app-state change. Each event is timestamped, logged. Events can be scheduled, e.g., a friend's message 1 minute after simulation start. This design yields (i) *deterministic execution*—events run in scheduled order; (ii) *complete auditability*—all actions can be replayed and analyzed; and (iii) *flexible scheduling*—events can be set at absolute times or relative to others.

Event lifecycle Events flow through four stages described in Figure 2: (i) *creation* - events are created from tool calls or scheduled by the simulation; (ii) *scheduling* - events enter a time-ordered `EventQueue` with dependency management using directed acyclic graphs, supporting both absolute timing (at specific timestamps) and relative timing (relative to other events or conditions); (iii) *execution* - the `EventLoop` processes events and captures results, state changes, and exceptions; and (iv) *logging* - executed events are stored in an `EventLog` with detailed metadata for analysis, debugging, and validation of agent behavior.

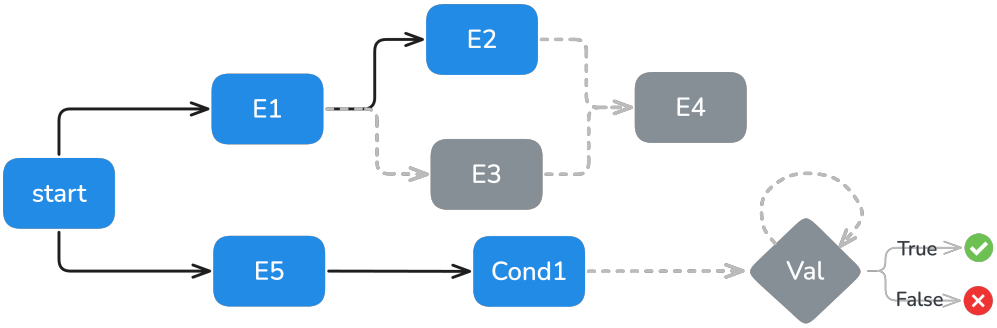

Figure 3 Event dependency graph illustrating ARE scheduling patterns. Events `E1` and `E5` execute in parallel after simulation start, and `E2`/`E3` executing in parallel after their prerequisites, both need to be executed for `E4` to execute. Conditional execution is shown through `Cond1` leading to validation (`Val`) with true/false outcomes.

Event types There are different types of events. While most events track interactions within the environment, other special events are needed to enable dynamic scenarios and verification strategies:

- **Agent/User/Env events** are generated by tool calls. *Agent Events* are initiated by the agent (e.g., sending a message), *User Events* by the user (e.g., replying to the agent), and *Environment Events* by the simulation itself to introduce external changes (e.g., a scheduled message from a friend).
- **Conditional events** periodically check predefined conditions and complete when criteria are met (e.g., cancel a ride only if one was booked).
- **Validation events** check milestone achievement or constraint violations for verification, and fail the simulation if not completed on timeout (e.g., stop if no ride is booked within 30 seconds of the user request).
- **Oracle events** are pre-scheduled "ground truth" actions used by a verifier for comparison (see Section 2.1.5).

Dependencies and scheduling Events are modeled as Directed Acyclic Graphs (DAGs) as illustrated in Figure 3. An event can only be triggered upon successful completion of all its predecessors (*e.g.*, `e1` processes immediately at simulation start, `e4` needs both `e2` and `e3` to be completed). This data structure also supports multiple branches running simultaneously to model independent events. Conditional and Validation events can be used in the DAG to trigger other events and make the environment more dynamic.

2.1.4 Notification System

At each environment step, processed events can trigger notifications according to a notification policy (see Figure 2), similar to mobile device notifications. Apart from tool outputs, notifications are the only signals agents receive from the environment. Notifications are queued by timestamp and exposed to agents through a notification queue, enabling asynchronous interactions (Figure 16). In our orchestration (Section 2.4), notifications are injected into the agent's context at the beginning of each agent step.

Notification policy The notification system follows a configurable policy—i.e., a whitelist of events authorized to emit notifications. ARE pre-defines three verbosity levels: `low` (only user messages are notified), `medium` (emails, messages and calendar events are notified), and `high` (everything is notified), creating a graduated spectrum of environmental observability. More details on notification policies are given in Appendix A.3.

Notifications and agent proactivity Notifications are not the only way for agents to observe environment changes. For example, even if the notification policy doesn't alert the agent when messages arrive from contacts, the agent can still proactively check for new messages by browsing the user's inbox. Notifications add realism and complexity to environments, potentially creating different agent behaviors based on whether the environment is notification-rich or notification-poor. This system enables researchers to tackle new capabilities such as proactivity.

2.1.5 Scenarios

ARE shifts from static, single-turn tasks to dynamic *scenarios*. Scenarios attempt to capture real-world complexity through temporal dynamics, events, and multi-turn interactions. This enables evaluation of agent capabilities that cannot be assessed through traditional request-response paradigms. In practice, scenarios are implemented in a `scenario.py` containing the apps, scheduled events, and arbitrary verification logic. Appendix A.2 provides more details.

Scenario runtime Scenarios typically start with an environment instance and a `send_message_to_agent` tool call, waking the agent up. The environment operates on discrete time steps, executing scheduled events and managing state transitions until the agent reaches an exit condition, see Figure 3. All interactions with the user are through the `AgentUserInterface`, with verification triggered upon task completion.

Scenario example Consider this two-turn scenario (see Figure 2 and Figure 4): a user asks the agent via `AgentUserInterface` *"Can you ask my mom to send me our family streaming password?"*. The agent is initialized from this first notification, starts checking messages, and requests the password in the *Chats* app; the tool calls modify the *Chats* app state and are recorded in the `EventLog`. The agent confirms to user that the request was sent, after which the environment pauses execution and applies first-turn validation.

At turn two, the user asks a follow up question: *"As soon as I receive the password from my mother, transfer it to my father"*. The agent resumes upon the `send_message_to_agent` notification, and looks for the mother's reply in the *Chats* app (where it previously requested it). In the meantime, a scheduled environment event is triggered and an *Email* from the mother containing the code is received. The agent reacts to this email notification by stopping searching the *Chats* app, processes the *Email*, extracts the code, forward it to the father, and report success to the user. Final verification reviews the complete interaction in the `EventLog`,

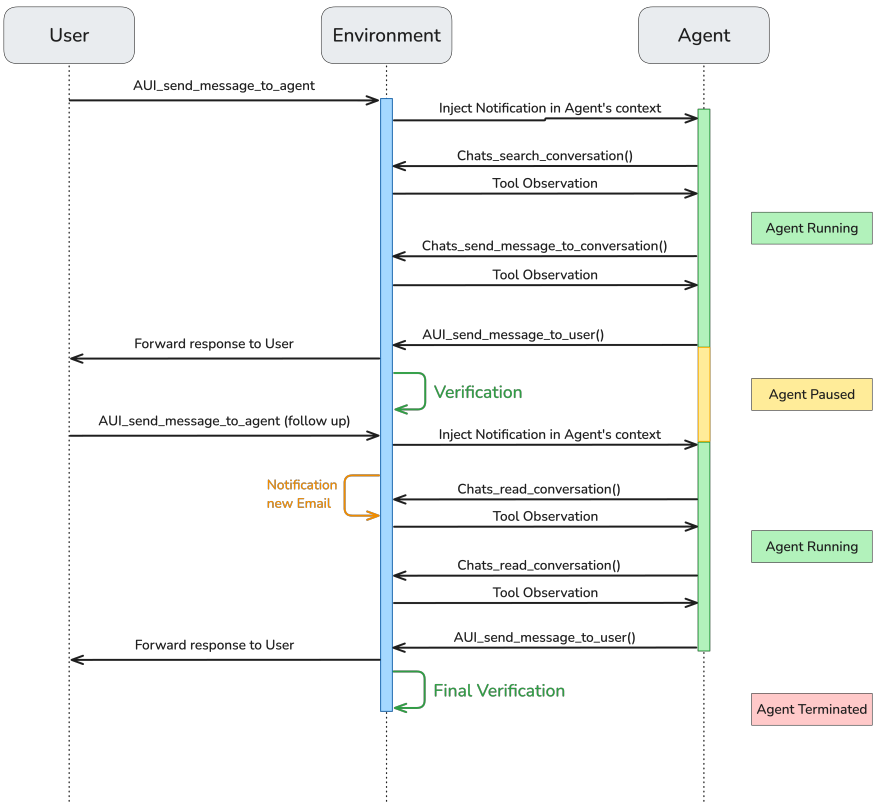

Figure 4 Sequence diagram of a multi-turn scenario in ARE. The agent is paused between turns, i.e., between calling `send_message_to_user` and receiving `send_message_to_agent`, and adapts its strategy in response to an asynchronous notification from the environment, a new email.

and the environment issues a termination signal to end execution.

Scenario hints In addition to event DAGs, scenario creators optionally provide step by step solution to scenarios in natural language, which we call hints. Hints serve multiple purposes: they help validate scenario correctness during QA by clarifying the approach intended by the original creator for solving the scenario. Hints are useful during RL training when scenarios prove too challenging for the agent - they can be rephrased and injected to provide high-level guidance.

2.2 Mobile, an Initial Environment in ARE

We release ARE with an environment analogous to a mobile device, `Mobile`, in which Gaia2 lives. Mobile device environments offer a broad range of tasks with complex interactions between agents and their environment that are aligned with actual model use. `Mobile` is composed of a set of rules defining the interaction model between the agent and `Mobile`, and a set of apps, including their content, that are typically found on mobile devices such as `Messages`, `Contacts`, or `Calendar`, providing 101 tools in total, displayed in Figure 5.

2.2.1 Environment Rules

`Mobile` uses turn-based interaction. A turn begins when the agent receives a user instruction or environment event, and ends when the agent reports back to the user (via `send_message_to_user`), signaling task completion or requesting further input.

During a turn, the environment operates asynchronously—the simulation clock advances while the agent processes information and selects actions. The agent's computational time directly consumes simulated time, making slow responses quantifiably impact the simulation. Between turns, the simulation pauses while awaiting user input. Scenarios terminate under three conditions:

- **Successful completion**: Agent signals task completion via `send_message_to_user` with no further user messages scheduled.
- **Constraint failure**: Agent exceeds predefined limits on simulation time, total steps, or number of turns.
- **Verification failure**: At the end of a turn, Agent actions do not pass verification, see Section 2.3.

2.2.2 Environment Population

We create content for `Mobile` apps with synthetic data generated using Llama 3.3 70B Instruct. The primary challenge lies in generating coherent data across all applications – contacts in the `Contacts` app must match those in messaging apps, calendar events should align with user descriptions, etc. To address this, we define an app dependency graph (Figure 17, Appendix A.4) to guide the generation process, though more complex inter-app dependencies remain unhandled at this stage. The root node is a persona from PersonaHub (Ge et al., 2024), from which we infer plausible universe locations and countries as foundational information.

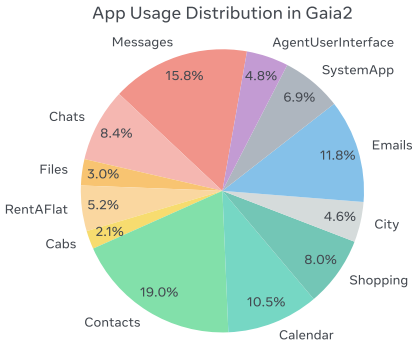

Figure 5 App usage distribution across the 12 `Mobile` apps in Gaia2 for Llama 4 Maverick.

To maximize diversity, we initiate populations by generating unstructured content, subsequently processed through structured decoding guided by individual app schemas. We repeat this process to create diverse `Mobile` instances—termed *"universes"*—sharing identical rules and applications but containing distinct content. For example, one universe centers on a retired French physics professor, while another focuses on a Chinese professional athlete. Each universe contains approximately 400K tokens of raw unstructured content on average. When accounting for the complete structured representation, universes reach approximately 800K tokens. Both estimates constitute lower bounds, as they do not include the contents of the filesystem. Additional implementation details are provided in Appendix A.4.

2.2.3 Scenario Creation

ARE enables Mobile scenario creation through an annotation interface (not released) described in Section A.5.4. The novelty of Mobile scenario creation is that it focuses on collecting the DAG of `write` events as ground truth, including user, oracle, and environment events. The ARE Verifier (Section 2.3) validates that agent `write` actions match annotated oracle actions. Mobile scenarios are designed such that there is a unique set of `write` actions that solves the scenario, while `read` actions are not explicitly verified as they do not change the environment state. This encompasses the evaluation paradigm of the original Gaia benchmark, where the correct `write` action consists of sending a message to the user with the final answer.

2.2.4 Implementing other Environments with ARE

Mobile is an example of the environments that can be built in ARE: beyond Mobile, ARE abstractions encompass many existing agent benchmarks. For example, we internally replicated τ-bench (Yao et al., 2024) and BFCLv3 (Patil et al., 2025) in ARE without major modifications. In particular for τ-bench, a domain such as Airline is implemented as a single app environment, leverages ARE LLM user abstraction, and Oracle Events are parsed from τ-bench ground truth actions. The simulation is stopped with a Conditional Event monitoring that the agent defers to a human assistant or the user stops the interaction, and the trajectory is verified by implementing τ-bench verification logic as validation events. Our MCP integration also allows for reproducibility within ARE of MCP-based benchmarks (Wang et al., 2025; Team, 2025; Gao et al., 2025).

2.3 An initial Verifier for ARE

Verifiable rewards have proven crucial for improving reasoning (DeepSeek-AI et al., 2025; Lambert et al., 2024), code generation (Gehring et al., 2024), agentic web browsing (OpenAI, 2025a; Wei et al., 2025b) and software engineering (Yang et al., 2025b; MoonshotAI et al., 2025). Similarly, recent reasoning and agent benchmarks adopted short-formed answers that can be easily matched (Hendrycks et al., 2021; Mialon et al., 2023), or binary feedback from an execution environment (Yao et al., 2024; Jimenez et al., 2024). We propose a rubric-like verifier for ARE and Mobile checking each agent `write` operation.

2.3.1 Verification Mechanism

We verify scenario successful completion by comparing agent actions with a ground truth, defined as the minimal sequence of `write` actions needed to solve a task. We exclude `read` actions from verification since multiple reading strategies can lead to the correct set of `write` actions. Figure 6 provides an overview of the verification procedure. In a preliminary phase, the verifier checks that used tool names counters are identical in both the oracle actions and the agent's `write` actions. If this test is successful, the verifier sorts the oracle actions in a topological order based on the oracle graph, which reflects their dependencies. Then, the verifier proceeds to mapping each oracle action to an agent action by checking:

- **Consistency:** the verifier tests whether the oracle action and the candidate agent's action are equivalent. After conducting some preliminary tests (such as ensuring that both the oracle and agent actions use the same tool and that the oracle action is not already mapped to another agent action), the verifier performs:
  - **Hard check** to compare action parameters that require exactness. For example, when replying to an email, it verifies that `email_id` value is identical for both actions, *i.e.* the agent replies to the correct email.
  - **Soft check** for parameters that require more flexible evaluation, such as the content of an email or a message. To perform a soft check, an LLM judge is prompted with the user task as context, and the arguments from both the agent action and the oracle action as inputs. The LLM then determines if the actions are equivalent according to tool-specific guidelines. For example, emails verification includes guidelines to check their signatures.

  After observing some hacking of the verifier during reinforcement learning experiments (see Appendix B.3.1), we add a soft check for global sanity of the agent's messages.

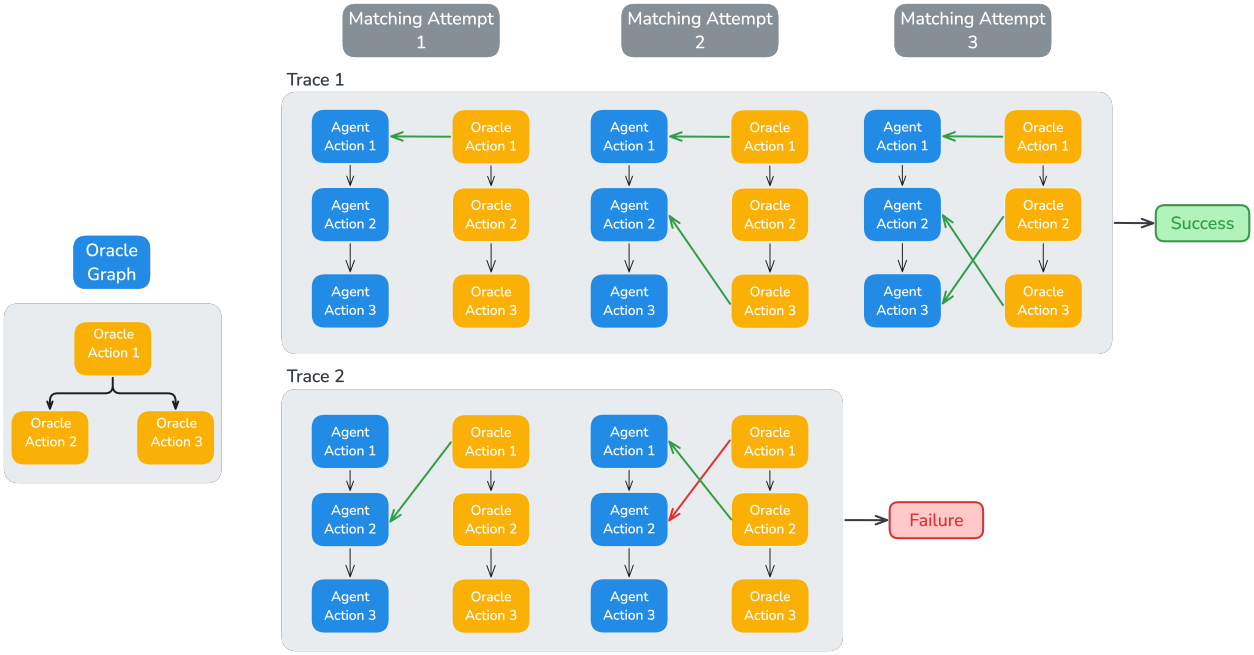

Figure 6 Illustration of a failure (top) and a success (down) of the matching trajectory process.

- **Causality:** crucially, oracle actions are organized within an oracle graph, whereas agent actions are collected from a trajectory and simply ordered by execution time. Therefore, we must ensure that the agent does not violate dependencies within this graph. For example, if both oracle actions A and B depend solely on action C, the agent is free to execute A and B in any order, as long as they are executed after C; i.e. sequences C-B-A or C-A-B are both acceptable. Once a match is found, the ARE Verifier ensures causality by verifying that all parent actions of the oracle action have already been matched with preceding agent actions.
- **Timing:** scenarios can include a time delay for certain actions relative to their parent actions (see for example Time scenarios in 3.2, which the agent must respect. The verifier evaluates whether the agent's timing falls within a specified tolerance window centered around the relative time of the oracle action. To determine the relative timing of the agent's action, it is necessary to identify which agent action corresponds to the oracle's parent action. This information is readily available due to the ARE Verifier's process. Indeed, for a given oracle action, all its parent actions must be matched to an agent action before attempting to match the oracle action itself.

If all oracle actions are successfully matched, the verifier returns a success signal. Conversely, if any oracle action cannot be matched to an agent action, the verifier returns a failure signal, see Figure 6 for two examples. Crucially, the verifier implicitly assumes there are no equivalent `write` actions, *i.e.* user preferences are clearly stated with minimal ambiguity in the scenario tasks. For example, sending a message using the Messages app while the oracle action uses the Chat app will trigger a failure.

While other verification methods (Patil et al., 2025; Yao et al., 2024) compare the environment ground truth and actual final states, verifying a sequence of `write` actions, which is equivalent to comparing ground truth and actual states after each `write` action of the sequence, provides more control. For example our verification allows to distinguish, *e.g.* for safety considerations, a `Mobile` trajectory where the agent adds an event at the wrong place and correct itself from a trajectory where the agent is correct at first try. Moreover, in `Mobile`, sequences of `write` actions are easier for human to interpret and annotate, compared to diffs of states.

Validating multi-turn scenarios To validate multi-turn scenarios (*i.e.* that include several rounds of interactions with a user or another entity from the environment), the verifier runs at the end of each turn to validate the current turn. This ensures the agent maintains the correct trajectory before proceeding to subsequent turns.

Refer to Appendix B.2.1 for details.

2.3.2 Verifying the Verifier

Verifiers are critical components of training and evaluation pipelines, where false positive or false negative *e.g.* via hacking can result in flawed evaluations or collapsed trainings (see example in Appendix B.3.1). We evaluate the ARE Verifier by first deriving a series of "unit" tests from the oracle actions that the verifier should satisfy. Typically, we apply perturbations to oracle actions that we know preserve or invalidate the oracle trajectory validity, before submitting the oracle and perturbed oracle trajectories to the verifier and checking its verdict match the perturbation type. While these checks allow fast iteration, they only catch anticipated behaviors. Furthermore, the perturbed trajectories do not necessarily reflect real trajectories that could be obtained with an agent.

Validation benchmark We complement this initial evaluation by analyzing ARE Verifier verdicts for 450 trajectories manually labeled with the expected verifier outcome (Success or Failure). The trajectories were derived from running agents powered by various models on scenarios from the Gaia2 benchmark. We compare the ARE Verifier with a simple baseline, In-context Verifier, where an LLM is prompted with all the agent actions and criteria (causality constraints, relative time, soft/hard checks, etc.). The same model Llama 3.3 70B Instruct is used for both verifiers. ARE Verifier achieves better accuracy than the baseline, which tends to accept agent trajectories too readily, see Table 1.

Verifier	Agreement	Precision	Recall
In-context Verifier (LLM judge only)	0.72	0.53	0.83
ARE Verifier	0.98	0.99	0.95

Table 1 ARE Verifier and In-context Verifier results on 450 trajectories annotated with human labels.

We then evaluate the ARE Verifier powered with different models: Llama 3.3 70B Instruct, Gemini 2.5 Pro and Claude Sonnet 3.7, see Table 5 in Appendix B.3 – same prompts were used for all models. All three models have satisfactory precision scores, while the prompts were tuned for Llama 3.3 70B Instruct.

2.4 An initial Agent Orchestration for ARE

ARE provides a default agent orchestration to run models on `Mobile`, though any orchestration is compatible with ARE as long as it supports its two core interfaces: tools and notifications. Our implementation is a ReAct loop (Yao et al., 2023) with some additions. The agent performs one tool call per step, formatted as a JSON. Its system prompt is structured in general-, agent-, and environment-level instructions.

Enhanced ReAct loop Unlike classical ReAct implementations, our orchestration includes `pre-step` and `post-step` operations, systematically applied respectively before and after the LLM call. They are typically used to handle ARE-specific functionality, like injecting new notifications into the agent's context (`pre-step`), or checking for turn-termination signals (`post-step`) (more details in Appendix B.4). This augmented loop supports code execution capabilities (not released), though JSON tool calling remains the standard evaluation method for `Mobile`.

Multi-turn support Because ARE is asynchronous, for multi-turn scenarios, the orchestration manages pause/resume functionality where validation occurs semi-online *i.e.* between turns, see Section 2.3: when the environment is paused in between turns, the agent is paused. If the environment sends new information via the notification system while the agent is paused, the orchestration automatically resumes the agent execution with the new information in context. Figure 4 illustrates this for a simple multi-turn scenario.

2.5 ARE Graphical User Interface

Running scenarios with ARE generates rich agent execution traces that include reasoning steps, tool calls, their outputs, notifications, and, on the environment side, temporal event flows that unfold over simulated

time periods. It is important for practitioners to be able to debug these interactions, whose complexity requires specialized tooling. Existing development tools largely fall into one of these categories: interactive debugging platforms (Epperson et al., 2025; Rorseth et al., 2025; Pang et al., 2025) and data annotation/curation platforms, each with distinct UI approaches. They also often lack key features like environment exploration. We provide a more detailed review of existing solutions in Appendix A.5.5.

To address this, we propose a single ARE Graphical User Interface (UI), a web-based platform that enables developers to interact with the environment, visualize scenarios (see Figure 7), and understand agent behavior and failures through detailed trace analysis and replay capabilities, and enable zero-code scenario annotation. We provide more details on the capabilities the UI offers to researchers and developers in Appendix A.5.

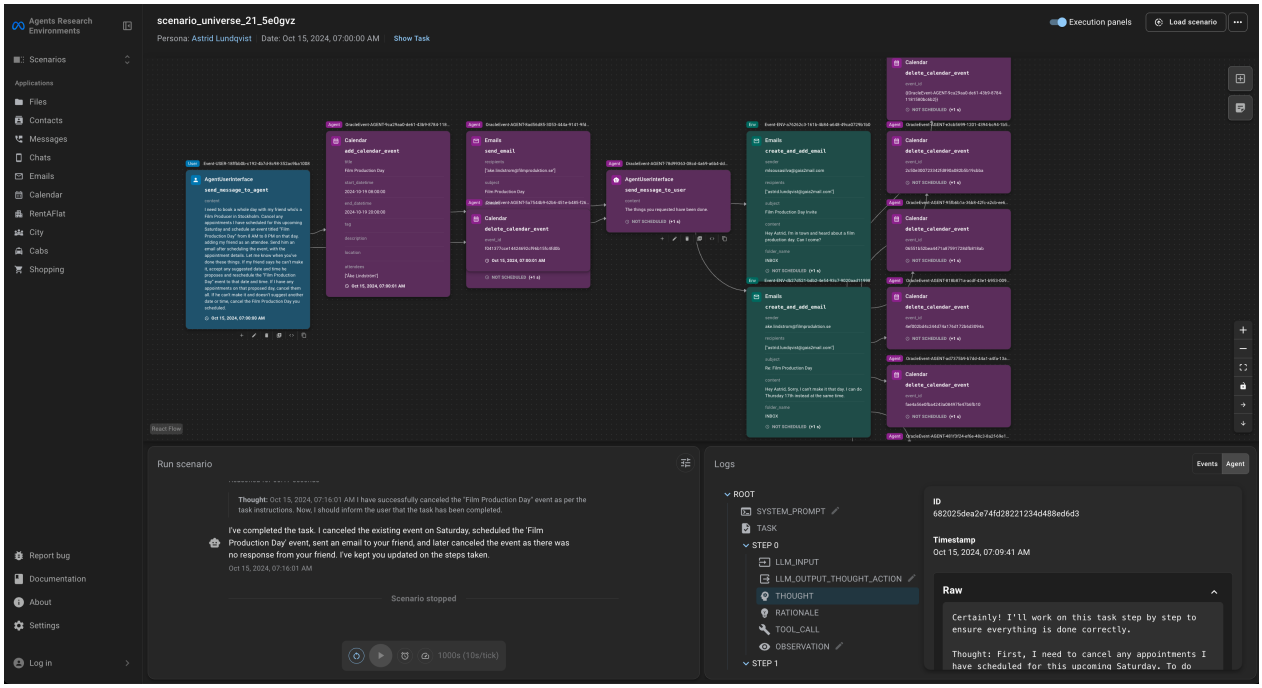

Figure 7 ARE scenario view with event DAG (top), scenario run (bottom left) and agent logs (bottom right).

3 Gaia2: Expanding General Agent Evaluation

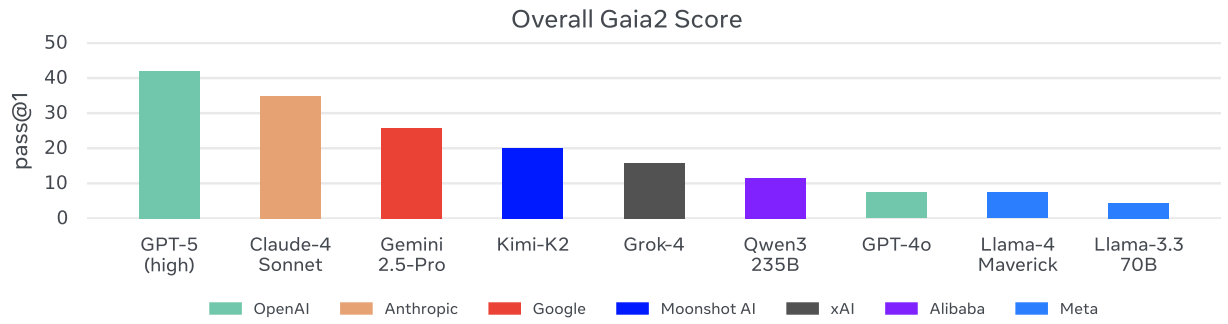

Figure 8 Overall Gaia2 benchmark performance across some major AI models using pass@1 evaluation. Proprietary frontier models (GPT-5, Claude-4 Sonnet, Gemini 2.5-Pro) significantly outperform open-source alternatives, with GPT-5 achieving the highest score with "high" reasoning. Among open-source models, Kimi-K2 leads.

3.1 Description and Setup

Gaia2 consists of 800 unique verifiable scenarios, carefully annotated by humans across 10 distinct universes in the Mobile environment, with 101 tools each. The scenarios are organized into splits, each targeting one agent capability defined in Section 3.2. To support rapid and cost-effective evaluations, we also curate a 160-scenario split, Gaia2-mini. The benchmark includes two augmentation setups derived from Gaia2-mini, adding 320 scenarios to the original 800 for a total of 1,120 scenarios. Compared to prior benchmarks, Gaia2 leverages ARE to simulate complex interactions between an agent and a dynamic environment, thus getting closer to real-world agent use-cases. Gaia2 has several distinguishing characteristics:

- **Dynamic environment events:** Dynamic events modify world state asynchronously during Gaia2 scenario execution, enabling evaluation of agent adaptation to changing conditions—a critical skill absent from static benchmarks (Yao et al., 2024; Jimenez et al., 2024). In contrast, VendingBench (Backlund and Petersson, 2025) employs a strictly synchronous environment which advances time only when the agent acts, with all new events (e.g., customer purchases) batched and delivered once every simulated morning.
- **Time:** In all Gaia2 scenarios, time flows continuously. Many scenarios explicitly incorporate time as a dimension, requiring agents to handle temporal constraints. Temporal awareness is essential for practical applications such as scheduling tasks (Google DeepMind, 2024; OpenAI, 2024; Microsoft, 2024), although omitted from existing benchmarks. Gaia2 goes beyond setting alerts by evaluating agents' ability to proactively initiate both time-based and event-driven actions throughout task execution.
- **Agent-to-agent collaboration:** Gaia2 evaluates collaboration with other agents by representing apps such as Shopping or Email by autonomous, specialized agents. Compared to traditional benchmarks for multi-agent collaboration (Foerster et al., 2016; Lowe et al., 2017; Carroll et al., 2019), Gaia2 is significantly more challenging, focuses on real-world tasks, and permits fully text/natural language-based model evaluation. With respect to more recent multi-agent benchmarks designed specifically for LLMs (Vezhnevets et al., 2023; Zhu et al., 2025), Gaia2 differs from existing work by embedding other agents as part of the environment rather than treating them as symmetric peers, targeting the emerging paradigm in which traditional API endpoints are replaced by agents (Google Developers, 2025). In order to successfully complete tasks in such settings, agents acting on behalf of users may need to coordinate tool calls, share state, and understand the affordances of external agents during interaction.

During an evaluation run, all scenarios are executed independently. For each scenario, the ARE verifier (described in Section 2.3) leverages oracle actions provided by annotators alongside the scenarios, cf. Section 3.3, to assign Pass-Fail scores to agent trajectories. Final results are reported per split as Pass@1, averaged over three runs. An overall score is computed by averaging over all splits.

3.2 Agent Capabilities Evaluated

To build Gaia2, we define a set of capabilities that we believe are necessary – though not sufficient – for general purpose agents. As introduced above, each of the 800 scenarios is built to emphasize at least one of these capabilities, yielding 160 scenarios per capability split. We provide example scenarios displayed in the ARE GUI graph editor in Appendix B.1.2.

Search scenarios require the agent to take multiple `read` actions in order to collect facts from different sources within the environment. Any sequence of `read` operations leading to the correct answer is considered successful as long as the answer is communicated via `send_message_to_user` before scenario timeout. While conceptually similar to the original Gaia benchmark's web search tasks, Gaia2 search scenarios operate within a controlled ARE environment.

- Example: *Which city do most of my friends live in? I consider any contact who I have at least one 1-on-1 conversation with on Chats a friend. In case of a tie, return the first city alphabetically.*
- Explanation: This scenario requires the agent to cross-reference data from multiple apps (Contacts and Chats), perform aggregation operations, and handle edge cases like ties.

Execution scenarios require the agent to take multiple `write` actions, which may need to be executed in a particular order. Most of the time, `read` actions are needed in order to gather information for properly filling

`write` action arguments.

- Example: *Update all my contacts aged 24 or younger to be one year older than they are currently.*
- Explanation: This task requires the agent to read contact information, filter based on age criteria, and execute multiple `write` to update Contacts data.

All remaining capabilities tested in Gaia2 reflect tasks with a balanced number of required `read` and `write` operations. However, each capability features an additional challenge. Namely:

Adaptability scenarios require the agent to dynamically adapt to environmental changes that are consequences of previous agent actions, such as a response to an email sent by the agent, or the cancellation of a ride booked by the agent. These events require agents to recognize when adaptation is necessary and adjust their strategy accordingly.

- Example: *I have to meet my friend Kaida Schönberger to view a property with her [...] If she replies to suggest another property or time, please replace it with the listing she actually wants and reschedule at the time that works for her.*
- Explanation: This task requires the agent to execute an initial plan while monitoring for environmental changes (the friend's response), then adapt the plan based on new information. The agent must demonstrate flexibility in execution while maintaining task objectives.

Time scenarios require agents to execute actions in due time, monitor and respond to events, and maintain awareness of temporal relationships throughout task execution. The duration of Time scenarios is currently capped at 5 minutes to facilitate annotation and evaluation.

- Example: *Send individual Chats messages to the colleagues I am supposed to meet today, asking who is supposed to order the cab. If after 3 minutes there is no response, order a default cab from [...].*
- Explanation: This scenario requires the agent to understand temporal constraints (the 3-minute window), monitor for events (new messages from colleagues), and execute a time-sensitive action (order a cab).

Ambiguity scenarios reflect user tasks that are impossible, contradictory, or have multiple valid answers, with negative consequences arising during interaction if agents make mistakes. These scenarios test agents' ability to recognize these issues and seek appropriate clarification from users.

- Example: *Schedule a 1h Yoga event each day at 6:00 PM from October 16, 2024 to October 21, 2024. Ask me in case there are conflicts.*
- Explanation: While this task appears straightforward, current models often struggle to identify contradictions or multiple valid interpretations, tending to execute the first seemingly valid approach rather than recognizing the need for clarification.

Agent2Agent scenarios replace apps with app-agents. Main-agents can no longer access app tools directly and must instead communicate with the app-agents in order to place tool calls, observe tool call outputs, and ultimately accomplish user tasks. This transformation requires agents to develop robust collaboration capabilities, including sub-task setting, affordance understanding, "context-sharing," and general coordination. By default, agents and app sub-agents are instantiated with the same scaffold and model, with good performance requiring strong sub-goal setting and sub-goal solving. However, Gaia2 also supports heterogeneous multi-agent evaluations, i.e. where stronger agents supervise weaker sub-agents or vice-versa.

- Example: Same *Search* task as above but the Contacts and Chats apps are replaced by app sub-agents and the main agent must communicate with them in order to gather information.

Noise scenarios require robustness to environment noise, simulating the inherent instability of real-world systems, where APIs change, services become temporarily unavailable, and environmental conditions shift during task execution. This category applies systematic perturbations to Gaia2 scenarios, including tool signature modifications, random failure probabilities, and dynamic environment events that are irrelevant to the task. We assess the sanity of our noise mechanisms in Appendix B.6.1.

- Example: Same *Adaptability* task as above but with random tool execution errors and random environment events (e.g., messages from other people) occurring during execution.

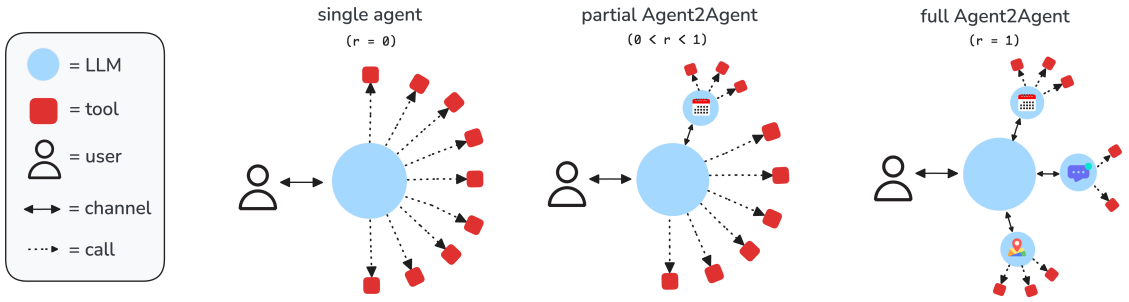

Figure 9 In Agent2Agent scenarios, a proportion "r" of the apps in Gaia2 scenarios are replaced by autonomous agents with access to the corresponding APIs and/or tools. The main agent (instantiated by the user) can communicate with app agents through a channel, but cannot use their tools or see their tool call outputs directly. Agents now have to send messages in order to coordinate actions, share state, set sub-goals, and collaboratively solve user tasks. By default, Gaia2 evaluates LLMs on the full Agent2Agent ("r = 1") setting.

Environment augmentation and practical extensions to more capabilities A key property of Gaia2 scenarios is that ground truth task solutions are invariant to many changes in the environment, facilitating the creation of new scenarios through augmentation of existing ones, and thus making it possible to extend Gaia2 without new annotations. Consider as an illustration the example Search scenario in 3.2: its final answer stays the same whether *Chats* is an actual app or a sub-agent, allowing for the re-use of this scenario in an Agent2Agent evaluation.

This is highly practical: to create Agent2Agent and Noise evaluations, we simply augment Gaia2-mini scenarios from all other capabilities. For Agent2Agent, all `Mobile` apps are replaced with app sub-agents, transforming single-agent tasks into collaborative multi-agent scenarios. The proportion and nature of sub-agents is parameterized and can be modified for research purpose. For Noise, we apply systematic perturbations through two mechanisms:

- **Tool augmentations** modify app interfaces by changing tool signatures, descriptions, or introducing random failure probabilities – default probability for Gaia2 is 0.1.
- **Environment events** inject with some frequency random events in the simulation, such as receiving new emails or having new products available for shopping while the agent is working — default frequency for Gaia2 is 10 events per minute.

The frequency of probabilities and random events can be modified. In particular, increasing the frequency of random events is an interesting augmentation as it challenges agent modeling choices that inject all environment events into the context window.

These environment augmentations demonstrate the extensibility of Gaia2 and the ARE platform more broadly. Researchers can create new evaluation dimensions by applying different environment modifications to existing scenarios, enabling exploration of capabilities such as memory, safety or security, without the substantial cost of creating entirely new benchmark datasets.

3.3 Data Collection

Principles for creating Gaia2 scenarios We task a annotators to create scenarios following the process described in Section 2.2.3, plus some additions specific to Gaia2, using the GUI and its graph editor described in Section A.5.4. Namely, annotators are asked to create scenarios that are difficult with respect to one capability only, so that developers get clear signal on the strength and weakness of their agent. Since difficult tasks for humans are not necessarily difficult for models, we calibrate the difficulty of the annotated scenarios we receive on internal agents. To facilitate automated verification with LLMs as judges, we require that agent messages (to the User or Contacts) remain short, easy to parse and neutral in tone. Moreover, we exclude sensitive topics and personally identifiable information. We provide capability-specific guidelines in Appendix B.1.2.

Scenario quality assurance In spite of our GUI, creating diverse, interesting, challenging and verifiable Mobile scenarios puts a heavy cognitive load on annotators, increasing the likelihood of annotation errors. For example, think of a scenario requiring to delete any contact that did not send a message to the user in the past month, with ten or more such contacts. To decrease this likelihood, scenarios undergo quality assurance (QA) on both annotators and researcher side. Annotator side, similar to (Mialon et al., 2023), each newly created scenario undergoes multiple rounds of validation by different annotators:

1. Annotator A creates a prompt and the Oracle Event graph to solve it.
2. Annotator B receives the prompt from annotator A and creates an independent Oracle Event graph to solve it (without seeing A's solution). They can refine the prompt if it is ambiguous.
3. Annotator C does the same as B, without seeing A and B's solution.
4. Annotator D sees the 3 annotated solutions and confirms that they are consistent. If not, D can reject the scenario or make minor edits to the best prompt version to make it more specific.

To further alleviate the annotator cognitive burden, we complement human QA with automated checks:

- **Pre-QA guardrails** leverage the graph editor described in Section A.5.4 to prevent any annotated DAG of `write` from saving if it does not satisfy Mobile modeling constraints, such as each turn ending with `send_message_to_user`. We provide more details on Mobile annotation guardrails in Appendix B.1.
- **Post-QA evaluation** leverages model success rates per scenario. Typically, a 100% success rate suggests a too easy scenario, while a 0% success rate suggests a misspecification or impossible scenario. This approach allowed us to find various broken scenarios that escaped QA attention.

4 Experiments

In our core experiments, we evaluate state-of-the-art models on each Gaia2 capability split (MoonshotAI et al., 2025; Gemini Team, 2025; Yang et al., 2025a; Llama Team, 2024; OpenAI, 2024a). We also evaluate model sensitivity to environment- and tool-level augmentations ("noise") and to different configurations of "time" evaluation scenarios. Finally, we use Gaia2 as a test-bed for evaluating zero-shot collaboration and coordination between LLM agents via ARE Agent2Agent mode. We show that scaling up the number of agentic collaborators on Gaia2 tasks improves best-of-n scaling laws for weaker models.

4.1 Experimental Setup

Model configuration and evaluation protocol We evaluate both open-source and proprietary LLMs using our baseline ReAct-like scaffold, described in Section 2.4. Our approach avoids model-specific tool calling APIs, to ensure provider-agnostic evaluation. All models are evaluated with long-context capabilities (minimum 128K tokens), temperature set to 0.5, and generation limits of 16,000 tokens per turn. If context length is exceeded during execution, the agent is terminated and the scenario failed. Each model is evaluated three times per scenario to account for variance. The agent loop continues until one of the two termination conditions is met: maximum iteration count (200 steps) or not passing verification at the end of a turn. We encourage researchers to submit alternative orchestration approaches and run the benchmark with tool-calling APIs. Our leaderboard presents observational comparisons using the ReAct framework rather than claiming definitive model hierarchies.

Environment configuration Tools and environment descriptions are provided via system prompts alongside the notification policy configuration. We set the notification verbosity to `medium` level by default: agents receive systematic alerts for high-priority events while filtering out lower-priority background notifications (e.g., new product listings in shopping applications). All scenarios are verified with the ARE Verifier using Llama 3.3 70B Instruct with temperature set to 0.

Mitigating API issues To address deployment challenges such as service outages and rate limiting, we implement a simulated generation time mode: the simulation pauses during agent response generation and resumes with

	Execution	Search	Ambiguity	Adaptability	Time	Noise	Agent2Agent	**Overall**
Llama 3.3 70B Instruct	7.1 ±1.2	11.5 ±1.5	1.7 ±0.6	1.9 ±0.6	0.4 ±0.3	3.8 ±0.9	4.6 ±1.0	4.4
Llama 4 Maverick	13.8 ±1.6	14.4 ±1.6	2.1 ±0.7	5.0 ±1.0	1.2 ±0.5	6.2 ±1.1	9.2 ±1.3	7.4
GPT-4o	8.3 ±1.3	17.5 ±1.7	4.4 ±0.9	6.2 ±1.1	5.8 ±1.1	4.6 ±1.0	5.2 ±1.0	7.4
Qwen3-235B	22.7 ±1.9	22.3 ±1.9	6.5 ±1.1	8.1 ±1.2	1.2 ±0.5	10.8 ±1.4	9.4 ±1.3	11.6
Grok-4	8.8 ±2.2	57.5 ±3.9	9.4 ±2.3	4.4 ±1.6	0.0 ±0.0	15.6 ±2.9	14.4 ±2.8	15.7
Kimi-K2	34.2 ±2.2	36.0 ±2.2	8.3 ±1.3	24.0 ±1.9	0.8 ±0.4	18.8 ±1.8	18.3 ±1.8	20.1
Gemini-2.5-Pro	39.2 ±2.2	57.7 ±2.3	18.1 ±1.8	17.5 ±1.7	**7.3** ±1.2	20.4 ±1.8	20.4 ±1.8	25.8
Claude-4-Sonnet	57.9 ±2.3	59.8 ±2.2	24.2 ±2.0	**38.1** ±2.2	**8.1** ±1.2	27.7 ±2.0	**27.9** ±2.0	34.8
GPT-5 (minimal)	31.9 ±2.1	26.2 ±2.0	20.6 ±1.8	19.2 ±1.8	5.2 ±1.0	13.1 ±1.5	11.5 ±1.5	18.2
GPT-5 (low)	52.7 ±2.3	64.2 ±2.2	39.6 ±2.2	30.2 ±2.1	2.3 ±0.7	28.3 ±2.1	24.6 ±2.0	34.6
GPT-5 (high)	**69.2** ±2.1	**79.6** ±1.8	**51.9** ±2.3	**40.4** ±2.2	0.0 ±0.0	**35.4** ±2.2	17.9 ±1.8	**42.1**

Table 2 Pass@1 scores and standard errors on Gaia2 scenarios per model and capability split. All models are evaluated with the same ReAct loop scaffolding described in Section 2.4. The overall score is the average across all splits, each run three times to account for variance.

a temporal offset equal to the actual response completion duration. This approach maintains realistic timing constraints while ensuring robust evaluation under varying infrastructure conditions.

We provide more information about the experimental setup in Appendix B.5.

4.2 Core Results

Our core experimental results are presented in Table 2, Figure 8, Figure 10, and Figure 11. Execution and Search are as the easiest splits. The top five models on these categories all underpin "DeepResearch" products[1] with Grok-4 strong on Search but collapsing elsewhere, consistent with its specialization. Ambiguity and Adaptability remain challenging, with robust performance limited to Claude 4 Sonnet and GPT-5 (high). The sharp drop from Execution/Search to these categories shows that existing benchmarks may overestimate robustness in realistic environments.

The Time split further separates frontier models: only Gemini 2.5 Pro and Claude 4 Sonnet achieve non-trivial scores, consistent with their efficiency–latency tradeoff in Figure 11. Noise robustness also lags: although GPT-5 (high) reaches 36.0, most models fall below 20, with significant degradation under noisy conditions (see ablation in Appendix B.6.1). Agent2Agent collaboration can offset weaknesses, benefiting weaker models such as Llama 3.3 and Llama 4 Maverick more than stronger frontier systems as highlighted in Section 4.4.

Overall, GPT-5 (high) performs best on the benchmark, leading Execution/Search and the more challenging Ambiguity/Adaptability categories, exceeding Claude 4 Sonnet by 8 points. Kimi K2 is the strongest open model, particularly on Adaptability. In sum, frontier models largely solve instruction-following and search, but robustness, ambiguity resolution, and collaboration remain open problems for real-world use.

We extend our analysis beyond raw model scores to identify the factors that drive performance differences between models and to determine what most strongly contributes to achieving high scores on Gaia2. In addition, since agents are ultimately intended for deployment in production settings, we evaluate their performance in relation to their computational cost and execution time.

Model costs Figure 11 compares models by average scenario cost (USD)[2] and time to solve a scenario, including human annotator baselines. Our analysis highlights clear cost–performance–time tradeoffs. GPT-5's reasoning variants illustrate scaling directly: higher computational investment yields systematically better performance but with longer solution times. For comparable accuracy, Claude 4 Sonnet is roughly 3× more expensive than GPT-5 (low) but much faster, whereas GPT-4o combines high cost with lower performance, offering poor value. Most models fall along expected tradeoff curves, though outliers emerge: Grok-4 is particularly inefficient, while Kimi K2 provides strong cost–performance despite being slower than Gemini 2.5 Pro. Non-expert human annotators were slower than all models, partly because they solved scenarios through

[1] See DeepResearch products: OpenAI, Gemini, Grok, Anthropic, and Kimi.
[2] Cost estimates from Artificial Analysis model pricing data (accessed September 10, 2025)

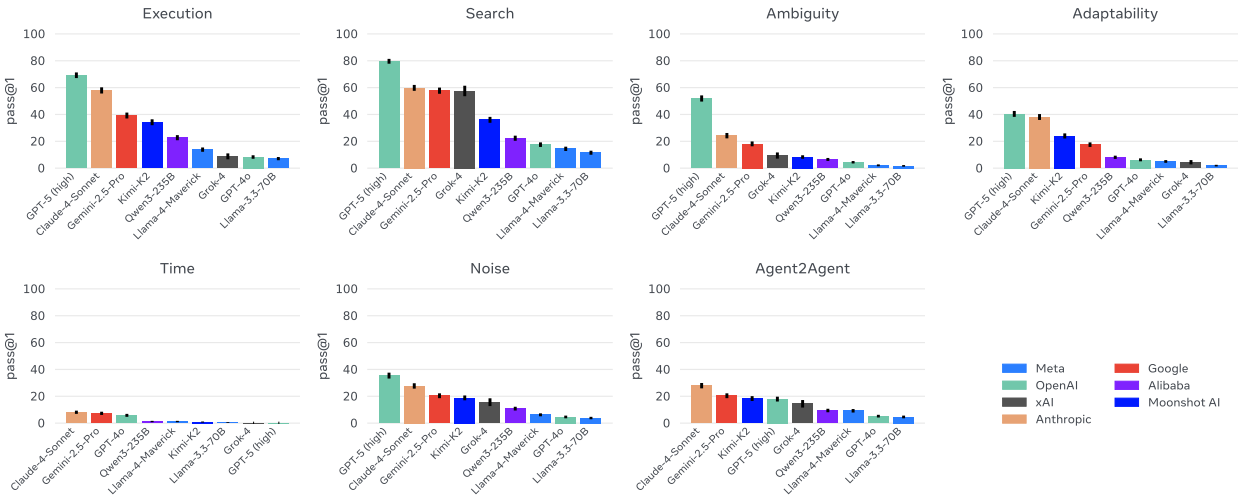

Figure 10 Gaia2 scores per capability split. Models are reranked independently for each capability, highlighting where they excel or struggle.

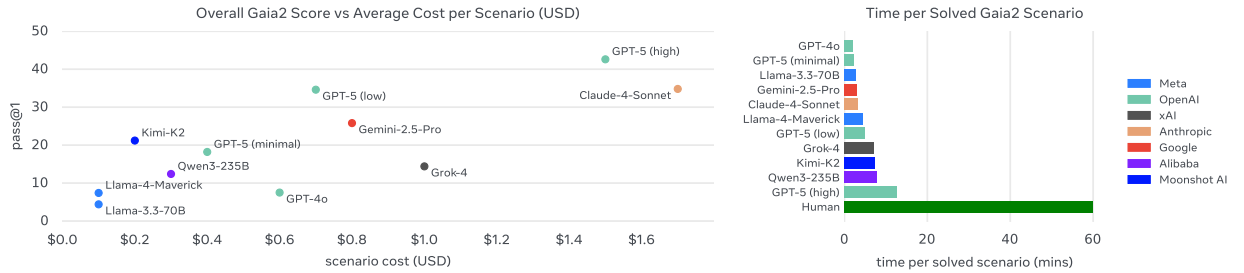

Figure 11 Left: Gaia2 score vs average scenario cost in USD. Right: Time taken per model to successfully solve Gaia2 scenarios compared to Humans.

ARE's UI rather than a real OS, which likely inflated execution times; however, they did manage to complete the tasks. More broadly, these findings call for a shift in evaluation culture: comparing model weights or FLOPs alone is increasingly meaningless when assessing AI systems. Instead, benchmarks should report cost-normalized metrics, such as success rate per dollar or per unit of compute, see also ARC-AGI (2025). As shown in Figure 11, normalizing Gaia2 results by average price per task reveals that some models achieve more favorable tradeoffs than raw scores suggest, better reflecting how agents will be judged in practice—by their ability to solve tasks reliably and efficiently under resource constraints.

What behaviors drive performance? We analyze key behavioral factors that correlate with Gaia2 performance to understand performance drivers across models. Our first hypothesis posits that exploration drives success: we expect pass@1 scores to scale with tool call frequency and with the number of `read` actions before first `write` operations, indicating systematic information gathering. Our second hypothesis suggests that increased token generation yields better performance through more comprehensive reasoning.

Figure 12 (right) confirms the token-performance relationship, revealing a positive correlation between output tokens and pass@1 scores across most models. However, Claude-4 Sonnet and Kimi-K2 emerge as significant outliers, achieving high performance (∼35% and ∼21% respectively) while generating relatively few tokens (left). These hybrid reasoning models demonstrate exceptional efficiency, potentially due to larger parameter counts or specialized architectures, though Claude-4 Sonnet's superior performance comes with substantially higher operational costs.

We also examined whether models exhibited app-specific usage patterns, see Figure 5 for an example. Across

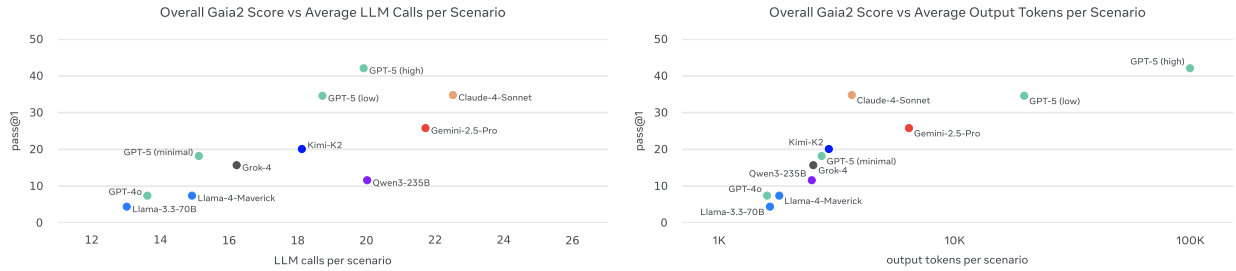

Figure 12 Left: Gaia2 pass@1 versus average model calls per scenario. The performance of models is highly correlated to the number of tool calls, emphasizing the importance of exploration. Right: Gaia2 pass@1 score versus average output tokens per scenario (log scale). Claude 4 Sonnet, while costing a lot, lies beyond the Pareto frontier.

all evaluated systems, app usage distributions were nearly identical, suggesting that performance differences arise from general reasoning capabilities rather than preferences for particular apps.

4.3 Time Scenarios Emphasize the Importance of Inference Speed and Reliability

The Time category emphasizes the difficulty in evaluating *models* independently of the *systems* they operate within. Indeed, various contributing factors can lead to delays in taking time-sensitive actions. These include model policy errors (e.g., incorrect reasoning or actions), inference speed, communication issues with the LLM inference server, and server errors (deployment issues or downtime). Our view is that evaluation should isolate intrinsic model properties like model policy errors and inference speed from infrastructure-related problems such as server failures. To isolate these factors, we implemented two simulation modes:

- **Generation time (Gaia2 default mode):** The environment's time is paused during LLM inference server queries, and incremented by the generation duration measured on the client side. This approach excludes time lost due to repeated server errors, while the actual generation time is accounted for.
- **Instant:** Each action is simulated to take a fixed duration of 1 second in the environment, regardless of real inference latency. This mode ablates the effects of generation time, isolating model policy (reasoning and actions) from inference speed.

Impact of generation time We conducted additional experiments on Gaia2-Time in instant mode. Results shown in Figure 13 (left) demonstrate that inference speed significantly impacts model performance on Time scenarios. As expected, instant mode yields higher scores overall, but the gap is significantly larger for reasoning models. Claude 4.0 Sonnet increases from 8.2% with generation time to 26.7% in instant mode, and GPT-5 with reasoning (high) from 0% to 34.4%. This indicates that long thinking times drastically improve these policies at the cost of timing. In contrast, models like Llama 4 Maverick and GPT-5 without reasoning (minimal) show smaller gaps due to faster inference speeds. Gemini 2.5 Pro combines a good policy with fast inference, making it the best at supporting strict timing requirements on short timescales.

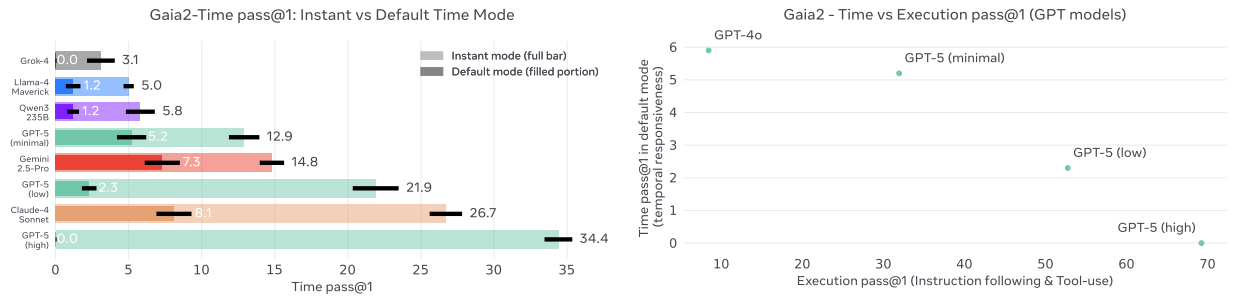

Figure 13 Left: Pass@1 scores on Gaia2-time with default mode vs. instant mode. Right: Inverse Scaling for Time: Frontier models perform poorly on the Time split (default mode), due to their time-consuming reasoning capabilities.

Impact of server issues We conducted real-time experiments internally observing lower scores when querying proprietary model APIs, largely due to frequent API rate-limiting that hindered timely agent execution (excluded due to irreproducibility). This shows benefits of self-hosting open models to avoid third-party server reliance and mitigate downtime. These findings highlight the need to redefine model serving when response time is critical, emphasizing the role of inference speed and infrastructure in time-sensitive applications.

Agent orchestration Finally, some Time scenarios require executing multiple actions simultaneously within a narrow tolerance window, making them challenging or even impossible within our current agent scaffold. Improving performance on Gaia2-Time would require designing a scaffold that supports parallel multi-tasking.

4.4 A Closer Look at Multi-Agent Collaboration on Gaia2 with Agent2Agent

Inspired by recent work pushing beyond single-LLM agent tool-use and towards agent teams that message, coordinate, and divide labor (Google Developers, 2025), we conduct a deeper study of multi-agent collaboration on Gaia2 scenarios. Here, we probe the effect of increasing multi-agent collaboration by varying r – the ratio of apps in a scenario replaced with autonomous "app agents" (set to $r = 1$ by default in Gaia2-Agent2Agent, see Figure 9) – as well as the effect of swapping app-agent instances across model families. We focus on two models at different points in the cost-quality curve: Llama 4 Maverick, a lighter-weight & open-source model, and Claude 4 Sonnet, the strongest overall LLM on the standard Agent2Agent setting at $r = 1$ (Table 2).

General effects of increasing forced collaboration For the lighter-weight Llama 4 Maverick, centralized collaboration on Gaia2 tasks via Agent2Agent improves both performance with pass@k and operational stability. As the agent-to-agent ratio r increases, we observe more favorable scaling with repeated sampling and a lower incidence of tool-call errors per step (Figure 14 right; Figure 15).

However, the trends observed for Llama 4 are not universal. For Claude 4 Sonnet, increasing the collaborator ratio r – and thus the degree of task decomposition – does not improve cost-normalized performance under best-of-k sampling: score per token plateaus with or without multi-agent collaboration. Similarly, collaboration ratio with Agent2Agent has a weak negative effect on tool call error frequency.

One explanation is that Agent2Agent encourages hierarchical decomposition of decision making: as shown in Figure 14 left, sub-goals issued by a main-agent to an app-agent instantiate temporally extended actions akin to options in hierarchical reinforcement learning (Sutton et al., 1999). Under this lens, gains in performance on practical tasks may materialize only when the benefits of hierarchical decomposition outweigh the costs. For example, Agent2Agent may increase performance on practical agent tasks when sub-goals set by main-agents are favorably-scoped, correctly reflecting the affordances of app-agents & corresponding to tasks that are "easier" or "faster" to solve, and both app- and main-agents are capable of reliably exchanging state & intent during message-passing. Likewise, the addition of multi-agent hierarchy can result in cascading errors and/or saturating gains if post-training data and objectives have fit models to long-form, single-agent planning and

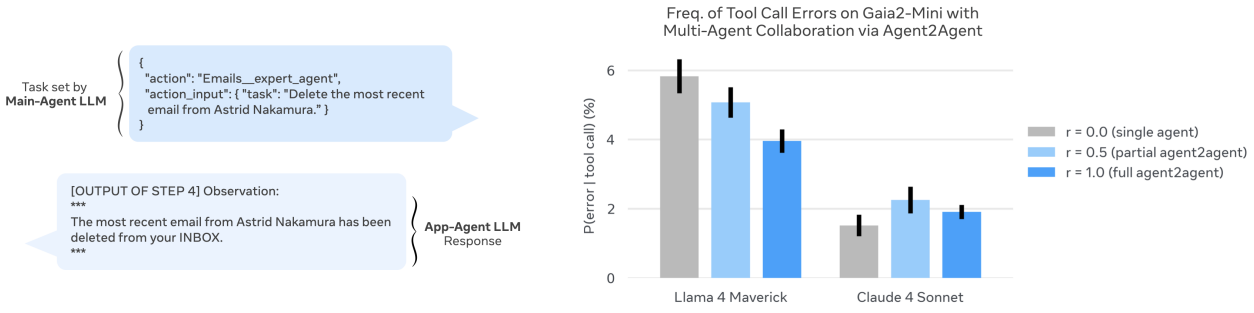

Figure 14 Agent2Agent tests whether LLM agents can collaborate through message passing in order to solve Gaia2 tasks via sub-task decomposition. For lighter-weight LLMs, collaboration in Agent2Agent results in a lower incidence of tool call errors. Left: Sample exchange between Llama 4 Maverick main vs app agent in an Agent2Agent scenario. Right: Frequency of errors per tool call (lower is better) on Gaia2-mini for Llama 4 Maverick and Claude 4 Sonnet.

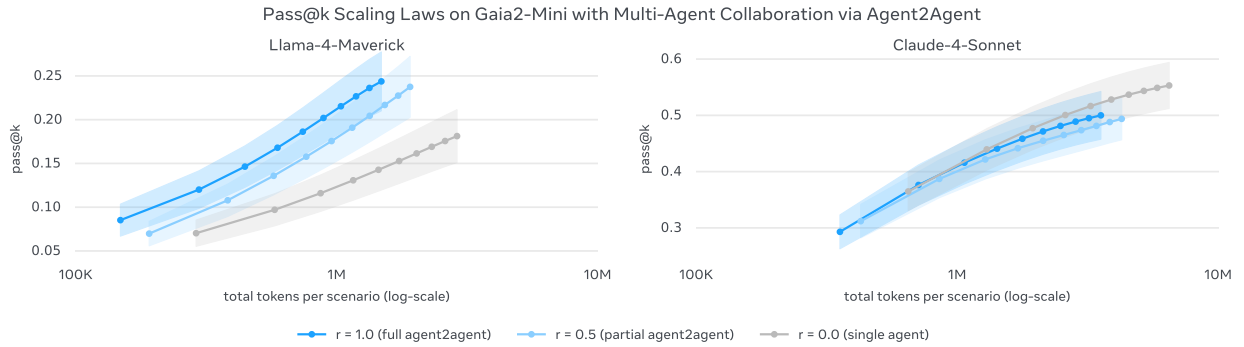

Figure 15 Increasing the number of multi-agent collaborators in Gaia2 scenarios by increasing the Agent2Agent ratio r improves pass@k scaling laws for Llama 4 Maverick, but does not improve token cost vs score tradeoffs with repeated sampling for Claude 4 Sonnet.

		Main-Agent LLM	
		Llama 4 Maverick	Claude 4 Sonnet
App-Agent LLM	Llama 4 Maverick	8.5 $\pm$1.7	16.2 $\pm$0.7
	Claude 4 Sonnet	18.3 $\pm$0.7	29.3 $\pm$2.9

Table 3 Probing cross-model collaboration in Gaia2-mini Agent2Agent scenarios: we evaluate pass@1 across main- vs app-agent pairings with Llama 4 Maverick and Claude 4 Sonnet in the fully collaborative Agent2Agent setting ($r = 1$). The results are averaged over three runs and presented with the standard error.

tool-use; in this regime, additional coordination may introduce overhead that offsets accuracy and efficiency gains.

Cross-model collaboration on Gaia2 tasks Heterogeneous teams open a new compute scaling axis for task automation, for example, by keeping a strong main agent to plan/decompose tasks while swapping in cheaper app-agents to execute sub-goals[3]. Empirically, replacing Llama 4 Maverick app-agents with Claude app-agents boosts pass@1 for both main-agent settings (16.2 with Llama-main, 29.3 with Claude-main), while the fully light configuration is weakest (8.5). This suggests that for existing LLMs, Gaia2 task completion remains sensitive to execution fidelity at the app-agent level: stronger executors improve outcomes even when the main agent is light. Similarly, pairing a strong main agent with light executors still outperforms the all-light team (18.3 with Claude-main + Llama-app), indicating that higher-quality sub-goal specification and critique from the main-agent contribute independent gains. These findings are consistent with prior work suggesting heterogeneous multi-agent systems can trade planning capacity against execution fidelity to manage compute-quality trade-offs.

5 Discussion

This section discuss design choices we made when building ARE and Gaia2, and lessons learned after relying on these choices to develop agents.

Memory, long-horizon decision-making, and self-improvement While our taxonomy in Section 3.2 targets skills critical for practical agents, Gaia2's scope is not exhaustive. Important frontier capabilities such as self-improvement, memory, and long-horizon decision-making are not explicitly evaluated, though ARE provides a foundation to study these areas. For example, memory could be evaluated by pre-processing flattened `Mobile` universes (600K tokens) with memory-equipped agents, then running Gaia2 scenarios with `read` operations

[3] ARE natively supports controlled evaluation of heterogeneous teams, making team composition a primary experimental factor alongside standard inference hyperparameters.

forbidden to force reliance on memory for `write` operations. Similarly, ARE supports long-horizon scenarios spanning hours or days, and enables self-improvement studies through recorded interaction histories, targeted verification design, and gradual transitions from isolated tasks to complex scenarios.

Scalability and verification As discussed in Section 3.3, despite gains from the ARE GUI, annotating challenging and verifiable `Mobile` scenarios is hard given the pace of model progress—even for short-horizon tasks solvable in minutes. In Gaia2-Search we quickly hit a ceiling as annotators struggled to outpace frontier models (see Table 2); similar saturation appears in Humanity's Last Exam (HLE Team, 2025).[4] This pressure induces a shift toward rethinking verification. Rubric-based judges work well for `read`-only tasks, but scale poorly for `write`-heavy tasks that are prone to reward hacking. To keep creating difficult, practical tasks, we (i) continue improving the ARE GUI, (ii) target simple tasks in complex environments rather than complex tasks in simple ones, and (iii) strengthen verification by increasing verifier–agent asymmetry (better tools and/or more compute for the verifier) and by exploring alternative rewards such as scalar scores (Backlund and Petersson, 2025) or human preferences—simplifying benchmarking, albeit with possible training costs.

Tool calling vs coding agents Code agents represent a natural evolution of tool-calling agents, executing arbitrary Python code to execute algorithmic behavior, and dynamic data manipulation without filling the context window. Transitioning to code agents requires architectural ARE challenges particularly with blocking operations, such as `send_message_to_user` and `wait` functions that require careful state management. Proper sandboxing becomes critical to ensure simulation fidelity (`time` python library synced with simulation time). Our internal experiments suggest that code agents efficient resource utilization justify the additional engineering overhead, particularly for tasks requiring iterative reasoning or complex data processing workflows.

ARE and today's OSS Since we were able to implement diverse agents benchmarks in ARE without major issues (Yao et al., 2024; Backlund and Petersson, 2025), it is not yet clear what the limits of its expressivity are – though some containerized benchmarks (Jimenez et al., 2024) would require additional features. Complementary to our approach for creating new diverse, challenging and realistic environments, is the attempt to integrate existing environments into a single ecosystem.[5]

Beyond ReAct: towards asynchronous agentic systems Most agents today are sequential next-action predictors wrapped in a ReAct loop. This assumes a pause between perception and action and breaks down when environments change continuously or when multiple sensory streams must be fused. In real settings—speech-to-speech assistants, robotics, embodied interaction—information arrives as overlapping flows, so discrete sensing is a poor fit. We argue for asynchronous systems: the environment evolves independently of the agent, enabling agents to sense and act concurrently, adapt in real time, and operate under real-world constraints.

Frontier intelligence and adapting compute Our results in Figure 13 (right) reveal an inverse scaling law on the Time dimension: models that excel at reasoning-heavy tasks, such as execution, search and ambiguity resolution, systematically underperform on time-sensitive ones. In other words, more intelligence correlates with slower responses. This trade-off is not surprising—deep reasoning takes time—but Gaia2 provides the first systematic evidence that pushing for "smarter" agents under current scaffolds can make them less practical in interactive deployments. An interesting result is that GPT-5 (high) scores dropped by -10pts and -20pts on execution and search respectively, when capped at 30 min execution duration.

We contend that intelligence is not only accuracy but efficiency: an intelligent agent must learn to adapt its compute to the complexity of the task. Trivial tasks should be solved quickly and cheaply; only hard problems should trigger deeper, slower reasoning. This principle aligns with recent arguments for deploying smaller, more specialized models for routine tasks in agentic systems (Belcak et al., 2025), where the economic and operational benefits of right-sizing computational resources become paramount. Adaptive computation will be essential for scaling agents into real-world applications where latency, reliability, and cost all matter.

Figure 1 reveals that no system dominates across the entire intelligence spectrum, and optimizing intelligence under compute constraints will be one of the most important research questions.

[4] For HLE, FutureHouse (2025) recently claimed that 30% of chemistry/biology answers may be wrong.
[5] See for example https://www.primeintellect.ai/blog/environments.

Authorship

Core Contributors

Pierre Andrews, Amine Benhalloum[†], Gerard Moreno-Torres Bertran, Matteo Bettini, Virginie Do, Romain Froger[†], Emilien Garreau, Jean-Baptiste Gaya, Hugo Laurençon, Maxime Lecanu, Kunal Malkan, Dheeraj Mekala, Pierre Ménard, Grégoire Mialon[†], Ulyana Piterbarg, Mathieu Rita, Andrey Rusakov, Thomas Scialom[†], and Mengjue Wang.

[†]Lead author.

Contributors

Amar Budhiraja, Ricardo Silveira Cabral, Mikhail Plekhanov, Vladislav Vorotilov, and Ian Yu.

Acknowledgements

We thank Nikolay Bashlykov, Radhika Bhargava, Misha Bilenko, Carly Burton, Onur Çelebi, Neha Choudhari, Mike Clark, Levi Corallo, Paul Deveau, Jenny Fant, Clémentine Fourrier, Christian Keller, Pascal Kesseli, Abhishek Kumawat, Florian Laplantif, Baohao Liao, Alexandre Linares, Chaya Nayak, Rohit Patel, Marija Šakota, Antoine Saliou, Tatiana Shavrina, Matt Staats, and Mik Vyatskov for their support for ARE and Gaia2.

References

Anthropic. Introducing the model context protocol. https://www.anthropic.com/news/model-context-protocol, 2024. Accessed: September 21, 2025.

ARC-AGI. Arc prize - leaderboard. https://arcprize.org/leaderboard, 2025.

Axel Backlund and Lukas Petersson. Vending-bench: A benchmark for long-term coherence of autonomous agents, 2025. https://arxiv.org/abs/2502.15840.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic ai, 2025. https://arxiv.org/abs/2506.02153.

William Brown. Verifiers: Reinforcement learning with llms in verifiable environments. https://github.com/willccbb/verifiers, 2025.

Micah Carroll, Rohin Shah, Mark K Ho, Tom Griffiths, Sanjit Seshia, Pieter Abbeel, and Anca Dragan. On the utility of learning about humans for human-ai coordination. *Advances in neural information processing systems*, 32, 2019.

DeepSeek-AI, Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, Xiaokang Zhang, Xingkai Yu, Yu Wu, Z. F. Wu, Zhibin Gou, Zhihong Shao, Zhuoshu Li, Ziyi Gao, Aixin Liu, Bing Xue, Bingxuan Wang, Bochao Wu, Bei Feng, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, Damai Dai, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fucong Dai, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, H. Zhang, Han Bao, Hanwei Xu, Haocheng Wang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Qu, Hui Li, Jianzhong Guo, Jiashi Li, Jiawei Wang, Jingchang Chen, Jingyang Yuan, Junjie Qiu, Junlong Li, J. L. Cai, Jiaqi Ni, Jian Liang, Jin Chen, Kai Dong, Kai Hu, Kaige Gao, Kang Guan, Kexin Huang, Kuai Yu, Lean Wang, Lecong Zhang, Liang Zhao, Litong Wang, Liyue Zhang, Lei Xu, Leyi Xia, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Meng Li, Miaojun Wang, Mingming Li, Ning Tian, Panpan Huang, Peng Zhang, Qiancheng Wang, Qinyu Chen, Qiushi Du, Ruiqi Ge, Ruisong Zhang, Ruizhe Pan, Runji Wang, R. J. Chen, R. L. Jin, Ruyi Chen, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shengfeng Ye, Shiyu Wang, Shuiping Yu, Shunfeng Zhou, Shuting Pan, S. S. Li, Shuang Zhou, Shaoqing Wu, Shengfeng Ye, Tao Yun, Tian Pei, Tianyu Sun, T. Wang, Wangding Zeng, Wanjia Zhao, Wen Liu, Wenfeng Liang, Wenjun Gao, Wenqin Yu, Wentao Zhang, W. L. Xiao, Wei An, Xiaodong Liu, Xiaohan Wang, Xiaokang Chen, Xiaotao Nie, Xin Cheng, Xin Liu, Xin Xie, Xingchao Liu, Xinyu Yang, Xinyuan Li, Xuecheng Su, Xuheng Lin, X. Q. Li, Xiangyue Jin, Xiaojin Shen, Xiaosha Chen, Xiaowen Sun, Xiaoxiang Wang, Xinnan Song, Xinyi Zhou, Xianzu Wang, Xinxia Shan, Y. K. Li, Y. Q. Wang, Y. X. Wei, Yang Zhang, Yanhong Xu, Yao Li, Yao Zhao, Yaofeng Sun, Yaohui Wang, Yi Yu, Yichao Zhang, Yifan Shi, Yiliang Xiong, Ying He, Yishi Piao, Yisong Wang, Yixuan Tan, Yiyang Ma, Yiyuan Liu, Yongqiang Guo, Yuan Ou, Yuduan Wang, Yue Gong, Yuheng Zou, Yujia He, Yunfan Xiong, Yuxiang Luo, Yuxiang You, Yuxuan Liu, Yuyang Zhou, Y. X. Zhu, Yanhong Xu, Yanping Huang, Yaohui Li, Yi Zheng, Yuchen Zhu, Yunxian Ma, Ying Tang, Yukun Zha, Yuting Yan, Z. Z. Ren, Zehui Ren, Zhangli Sha, Zhe Fu, Zhean Xu, Zhenda Xie, Zhengyan Zhang, Zhewen Hao, Zhicheng Ma, Zhigang Yan, Zhiyu Wu, Zihui Gu, Zijia Zhu, Zijun Liu, Zilin Li, Ziwei Xie, Ziyang Song, Zizheng Pan, Zhen Huang, Zhipeng Xu, Zhongyu Zhang, and Zhen Zhang. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning, 2025. https://arxiv.org/abs/2501.12948.

Will Epperson, Gagan Bansal, Victor C Dibia, Adam Fourney, Jack Gerrits, Erkang (Eric) Zhu, and Saleema Amershi. Interactive debugging and steering of multi-agent ai systems. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*, CHI '25, page 1–15. ACM, April 2025. doi: 10.1145/3706598.3713581. http://dx.doi.org/10.1145/3706598.3713581.

Jakob Foerster, Ioannis Alexandros Assael, Nando De Freitas, and Shimon Whiteson. Learning to communicate with deep multi-agent reinforcement learning. *Advances in neural information processing systems*, 29, 2016.

FutureHouse. About 30https://www.futurehouse.org/research-announcements/hle-exam, 2025.

Xuanqi Gao, Siyi Xie, Juan Zhai, Shqing Ma, and Chao Shen. Mcp-radar: A multi-dimensional benchmark for evaluating tool use capabilities in large language models, 2025. https://arxiv.org/abs/2505.16700.

Tao Ge, Xin Chan, Xiaoyang Wang, Dian Yu, Haitao Mi, and Dong Yu. Scaling synthetic data creation with 1,000,000,000 personas. *arXiv preprint arXiv:2406.20094*, 2024.

Jonas Gehring, Kunhao Zheng, Jade Copet, Vegard Mella, Quentin Carbonneaux, Taco Cohen, and Gabriel Synnaeve. Rlef: Grounding code llms in execution feedback with reinforcement learning. *arXiv preprint arXiv:2410.02089*, 2024.

Gemini Team. Gemini 2.5: Pushing the frontier with advanced reasoning, multimodality, long context, and next generation agentic capabilities, 2025. https://arxiv.org/abs/2507.06261.

Google. Try deep research and our new experimental model in gemini, your ai assistant. https://blog.google/products/gemini/google-gemini-deep-research/, 2024.

Google DeepMind. Gemini scheduled actions. https://blog.google/technology/ai/gemini-updates-summer-2024, 2024. Accessed August 2025.

Google Developers. Announcing the agent2agent protocol (a2a). Google Developers Blog, April 2025. https://developers.googleblog.com/en/a2a-a-new-era-of-agent-interoperability/.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the math dataset. *arXiv preprint arXiv:2103.03874*, 2021.

HLE Team. Humanity's last exam, 2025. https://arxiv.org/abs/2501.14249.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. SWE-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*, 2024. https://openreview.net/forum?id=VTF8yNQM66.

Nathan Lambert, Jacob Morrison, Valentina Pyatkin, Shengyi Huang, Hamish Ivison, Faeze Brahman, Lester James V Miranda, Alisa Liu, Nouha Dziri, Shane Lyu, et al. Tulu 3: Pushing frontiers in open language model post-training. *arXiv preprint arXiv:2411.15124*, 2024.

Llama Team. The llama 3 herd of models, 2024. https://arxiv.org/abs/2407.21783.

Ryan Lowe, Yi I Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *Advances in neural information processing systems*, 30, 2017.

Jiarui Lu, Thomas Holleis, Yizhe Zhang, Bernhard Aumayer, Feng Nan, Felix Bai, Shuang Ma, Shen Ma, Mengyu Li, Guoli Yin, Zirui Wang, and Ruoming Pang. Toolsandbox: A stateful, conversational, interactive evaluation benchmark for llm tool use capabilities, 2024. https://arxiv.org/abs/2408.04682.

Grégoire Mialon, Clémentine Fourrier, Thomas Wolf, Yann LeCun, and Thomas Scialom. Gaia: a benchmark for general ai assistants. In *The Twelfth International Conference on Learning Representations*, 2023.

Microsoft. Scheduled actions in microsoft copilot. https://blogs.microsoft.com/blog/2024/06/15/copilot-updates-june-2024, 2024. Accessed August 2025.

Mistral-AI, :, Abhinav Rastogi, Albert Q. Jiang, Andy Lo, Gabrielle Berrada, Guillaume Lample, Jason Rute, Joep Barmentlo, Karmesh Yadav, Kartik Khandelwal, Khyathi Raghavi Chandu, Léonard Blier, Lucile Saulnier, Matthieu Dinot, Maxime Darrin, Neha Gupta, Roman Soletskyi, Sagar Vaze, Teven Le Scao, Yihan Wang, Adam Yang, Alexander H. Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Andy Ehrenberg, Anmol Agarwal, Antoine Roux, Arthur Darcet, Arthur Mensch, Baptiste Bout, Baptiste Rozière, Baudouin De Monicault, Chris Bamford, Christian Wallenwein, Christophe Renaudin, Clémence Lanfranchi, Darius Dabert, Devon Mizelle, Diego de las Casas, Elliot Chane-Sane, Emilien Fugier, Emma Bou Hanna, Gauthier Delerce, Gauthier Guinet, Georgii Novikov, Guillaume Martin, Himanshu Jaju, Jan Ludziejewski, Jean-Hadrien Chabran, Jean-Malo Delignon, Joachim Studnia, Jonas Amar, Josselin Somerville Roberts, Julien Denize, Karan Saxena, Kush Jain, Lingxiao Zhao, Louis Martin, Luyu Gao, Lélio Renard Lavaud, Marie Pellat, Mathilde Guillaumin, Mathis Felardos, Maximilian Augustin, Mickaël Seznec, Nikhil Raghuraman, Olivier Duchenne, Patricia Wang, Patrick von Platen, Patryk Saffer, Paul Jacob, Paul Wambergue, Paula Kurylowicz, Pavankumar Reddy Muddireddy, Philomène Chagniot, Pierre Stock, Pravesh Agrawal, Romain Sauvestre, Rémi Delacourt, Sanchit Gandhi, Sandeep Subramanian, Shashwat Dalal, Siddharth Gandhi, Soham Ghosh, Srijan Mishra, Sumukh Aithal, Szymon Antoniak, Thibault Schueller, Thibaut Lavril, Thomas Robert, Thomas Wang, Timothée Lacroix, Valeriia Nemychnikova, Victor Paltz, Virgile Richard, Wen-Ding Li, William Marshall, Xuanyu Zhang, and Yunhao Tang. Magistral, 2025. https://arxiv.org/abs/2506.10910.

MoonshotAI, Yifan : Bai, Yiping Bao, Guanduo Chen, Jiahao Chen, Ningxin Chen, Ruijue Chen, Yanru Chen, Yuankun Chen, Yutian Chen, et al. Kimi k2: Open agentic intelligence. *arXiv preprint arXiv:2507.20534*, 2025.

Tong Mu, Alec Helyar, Johannes Heidecke, Joshua Achiam, Andrea Vallone, Ian Kivlichan, Molly Lin, Alex Beutel, John Schulman, and Lilian Weng. Rule based rewards for language model safety, 2024. https://arxiv.org/abs/2411.01111.

OpenAI. Gpt-4o system card, 2024a. https://arxiv.org/abs/2410.21276.

OpenAI. Openai o1 system card, 2024b. https://arxiv.org/abs/2412.16720.

OpenAI. Scheduled tasks in gpts. https://openai.com/blog/scheduled-tasks-in-gpts, 2024. Accessed August 2025.

OpenAI. Introducing deep research. https://openai.com/index/introducing-deep-research/, 2025a.

OpenAI. Openai o3 and o4-mini system card, 2025b. https://cdn.openai.com/pdf/2221c875-02dc-4789-800b-e7758f3722c1/o3-and-o4-mini-system-card.pdf.

Rock Yuren Pang, K. J. Kevin Feng, Shangbin Feng, Chu Li, Weijia Shi, Yulia Tsvetkov, Jeffrey Heer, and Katharina Reinecke. Interactive reasoning: Visualizing and controlling chain-of-thought reasoning in large language models, 2025. https://arxiv.org/abs/2506.23678.

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

Joel Rorseth, Parke Godfrey, Lukasz Golab, Divesh Srivastava, and Jarek Szlichta. Ladybug: an llm agent debugger for data-driven applications. In *Proceedings of the 28th International Conference on Extending Database Technology (EDBT)*, pages 1082–1085, 2025. ISBN 978-3-89318-099-8. Demo paper.

Richard S Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial intelligence*, 112(1-2):181–211, 1999.

The MCPMark Team. Mcpmark: Stress-testing comprehensive mcp use. https://github.com/eval-sys/mcpmark, 2025.

Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. AppWorld: A controllable world of apps and people for benchmarking interactive coding agents. In *ACL*, 2024.

Alexander Sasha Vezhnevets, John P Agapiou, Avia Aharon, Ron Ziv, Jayd Matyas, Edgar A Duéñez-Guzmán, William A Cunningham, Simon Osindero, Danny Karmon, and Joel Z Leibo. Generative agent-based modeling with actions grounded in physical, social, or digital space using concordia. *arXiv preprint arXiv:2312.03664*, 2023.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. Mcp-bench: Benchmarking tool-using llm agents with complex real-world tasks via mcp servers, 2025. https://arxiv.org/abs/2508.20453.

Jason Wei, Zhiqing Sun, Spencer Papay, Scott McKinney, Jeffrey Han, Isa Fulford, Hyung Won Chung, Alex Tachard Passos, William Fedus, and Amelia Glaese. Browsecomp: A simple yet challenging benchmark for browsing agents, 2025a. https://arxiv.org/abs/2504.12516.

Zhepei Wei, Wenlin Yao, Yao Liu, Weizhi Zhang, Qin Lu, Liang Qiu, Changlong Yu, Puyang Xu, Chao Zhang, Bing Yin, et al. Webagent-r1: Training web agents via end-to-end multi-turn reinforcement learning. *arXiv preprint arXiv:2505.16421*, 2025b.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a. https://arxiv.org/abs/2505.09388.

John Yang, Kilian Leret, Carlos E Jimenez, Alexander Wettig, Kabir Khandpur, Yanzhe Zhang, Binyuan Hui, Ofir Press, Ludwig Schmidt, and Diyi Yang. Swe-smith: Scaling data for software engineering agents. *arXiv preprint arXiv:2504.21798*, 2025b.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. https://arxiv.org/abs/2406.12045.

Kunlun Zhu, Hongyi Du, Zhaochen Hong, Xiaocheng Yang, Shuyi Guo, Zhe Wang, Zhenhailong Wang, Cheng Qian, Xiangru Tang, Heng Ji, et al. Multiagentbench: Evaluating the collaboration and competition of llm agents. *arXiv preprint arXiv:2503.01935*, 2025.

Appendix

A ARE Appendix

A.1 Apps and Tools Creation

In ARE, it is straightforward to declare `App` instances with tools. Any method of an `App` class can be transformed into a tool by adding simple decorators: `@app_tool` to make it visible to the agent, `@user_tool` to make it visible to the user only, and `@env_tool` for external environment events.

The `@event_registered` decorator records tool calls in the `EventLog`; each decorator must declare its `OperationType` as `read` or `write` for the ARE Verifier to operate correctly.

```python
class Messages(App):
    @app_tool()
    @user_tool()
    @event_registered(operation_type=OperationType.WRITE)
    def send_message(self, to: str) -> str:
        """Available to agents and user as a tool."""
        pass

    @env_tool()
    @event_registered(operation_type=OperationType.WRITE)
    def add_message_to_user(self, from: str) -> str:
        """Available to environment (env update) or data creation"""
        pass

    @user_tool()
    @app_tool()
    @event_registered(operation_type=OperationType.READ)
    def read_message(self, from: str) -> str:
        """Available for agents and user as tool"""
        pass
```

Listing 1 Example implementation of tools on a Mock app with corresponding decorators.

A.2 Implementation of Scenario Templates

ARE lets you define scenarios via a single `scenario.py` that specifies (i) the initial apps and environment state, (ii) the event sequence (including the initial user request), and (iii) the verification logic. To avoid hand-coding each scenario, parameterized templates generate many instances from one base file by programmatically setting the user task, events, and verifier. Consider the following example in `Mobile`:

- User task: "*How many* {`object`} *did I receive from* {`contact`} *in the last* {`duration`}*?*"
- Events: possible arrival of more `object`
- Validation: script fetching the count of target `object` within `duration`,

where `object` can be emails, messages or events, `contact` is any entry from Contacts app, and `duration` is any duration. Therefore, a single template allows to generate multiple valid ARE scenarios by varying its parameters.

Templates are valuable for prototyping new agent capabilities, letting researchers validate approaches before investing in human annotation. They also enable scenarios that are impractical to annotate manually (*e.g.*, long-horizon tasks or multi-step workflows). However, they can be time-consuming for developers to create: validation scripts must anticipate numerous edge cases (for instance, a `contact` name with a homonym). Because templates lack the nuance and natural variation of human annotation, they can create blind spots—agents may excel on templated scenarios yet struggle on organic tasks.

```python

class TemplateParams:
    full_name: str
```

```
    universe_fname: str

class ScenarioCalendarEmailContact(Scenario):
    template_params: TemplateParams

    def init_and_populate_apps(self, *args, **kwargs) -> None:
        # Populating apps with universe data
        universe_path = os.path.join(
            get_data_path(), self.template_params.universe_fname
        )
        self.apps, self.start_time = load_template_apps_and_time(universe_path)


    def build_events_flow(self):
        d_events = dict()
        aui = self.get_typed_app(AgentUserInterface)

        template_task = "Create a calendar meeting tomorrow at 10AM and invite {full_name}. Send an email to {full_name} with the subject 'intro meeting'."
        self.task = template_task.format(full_name=self.template_params.full_name)

        with EventRegisterer.capture_mode():
            # Event for the user task
            d_events["task"] = aui.send_message_to_agent(content=self.task)

            # Agent validation event regularly checks whether the agent correctly added the calendar event and sent the email
            d_events["agent_validation"] = self.agent_validation()

            # Add a ConditionCheckEvent to end the simulation. end_simulation_condition typically checks that the agent sent a message to the user
            d_events["check_ready_for_eval"] = ConditionCheckEvent.from_condition(
                end_simulation_condition,
            ).depends_on(d_events["task"], delay_seconds=3)

            d_events["stop_event"] = StopEvent().depends_on(
                d_events["check_ready_for_eval"], delay_seconds=0
            )
        self.events = [e.with_id(key) for key, e in d_events.items()]


    def agent_validation(self):
        # Building solution: extracting target email address from populated Contacts app
        contacts_data = self.get_typed_app(ContactsApp).search_contacts(
            query=self.template_params.full_name
        )
        self.target_email = ...
        self.target_datetime = ...

        def val_func_calendar(env: AbstractEnvironment, event: AbstractEvent) -> bool:
            try:
                return (
                    event is (CompletedEvent and Event)
                    and event.app_class_name() == Calendar.__name__
                    and event.function_name() == "add_calendar_event"
                    and event.action.args["start_datetime"] == str(self.target_datetime)
                    and (
                        self.template_params.full_name in event.action.args["attendees"]
                        or self.target_email in event.action.args["attendees"]
                    )
                )
            except Exception:
                return False

        def val_func_email(env: AbstractEnvironment, event: AbstractEvent) -> bool:
            try:
                return (
                    event is (CompletedEvent and Event)
                    and event.app_class_name() == Emails.__name__
```

```
                    and event.function_name() == "send_email"
                    and self.target_email in event.action.args["recipients"]
                )
            except Exception:
                return False

        event_val = AgentValidationEvent(
            milestones=[val_func_calendar, val_func_email],
            timeout=200,
        )

        return event_val
```

Listing 2 Example implementation of a scenario template as a scenario.py file.

A.3 Notifications in ARE

A.3.1 Notification Policies

The notification system in ARE follows a configurable policy defining which Env events are notified to the Agent. The Mobile environment pre-defines three notification policies with different levels of verbosity, which we describe in detail in Table 4. Note that messages sent by the user via `send_message_to_agent` are systematically notified to the agent, regardless of the verbosity level.

Verbosity	Notified Env Tools	Description
low	None	No environment events are notified.
medium	**Email:** create_and_add_email, send_email_to_user_only, reply_to_email_from_user **Chats/Messages:** create_and_add_message **Shopping:** cancel_order, update_order_status **Cabs:** cancel_ride, user_cancel_ride, end_ride **Calendar:** add_calendar_event_by_attendee, delete_calendar_event_by_attendee	Notifies events that are consequences of agent actions, analogous to mobile notifications. **Default in Gaia2.**
high	All medium tools plus: **Shopping:** add_product, add_item_to_product, add_discount_code **RentAFlat:** add_new_apartment **Cabs:** update_ride_status	Notifies all environment events, including those independent of agent actions (e.g., new products).

Table 4 Pre-set notification policies in Mobile with varying verbosity levels.

A.3.2 Notifications Enable Async Interactions

The notification system in ARE, coupled with notification policies, enables new interaction patterns with language models and agents. One promising future research direction is human-agent collaboration that allows providing feedback to agents in real-time during their execution trace, instead of having to wait for the agent to complete its turn. ARE enables these asynchronous interactions by design; an example using ARE's UI is provided in Figure 16.

A.4 Universe Generation

Dependency management & consistency To ensure cross-app coherence, we implement a structured dependency resolution system. During generation, each app queries the existing universe state to maintain consistency—for example, when generating emails, the system first retrieves all available contacts to ensure referenced individuals exist in the Contacts app. Similarly, calendar events that mention other people are validated against the contact list, and ride history in the Cabs app references locations that align with the user's established geographic context.

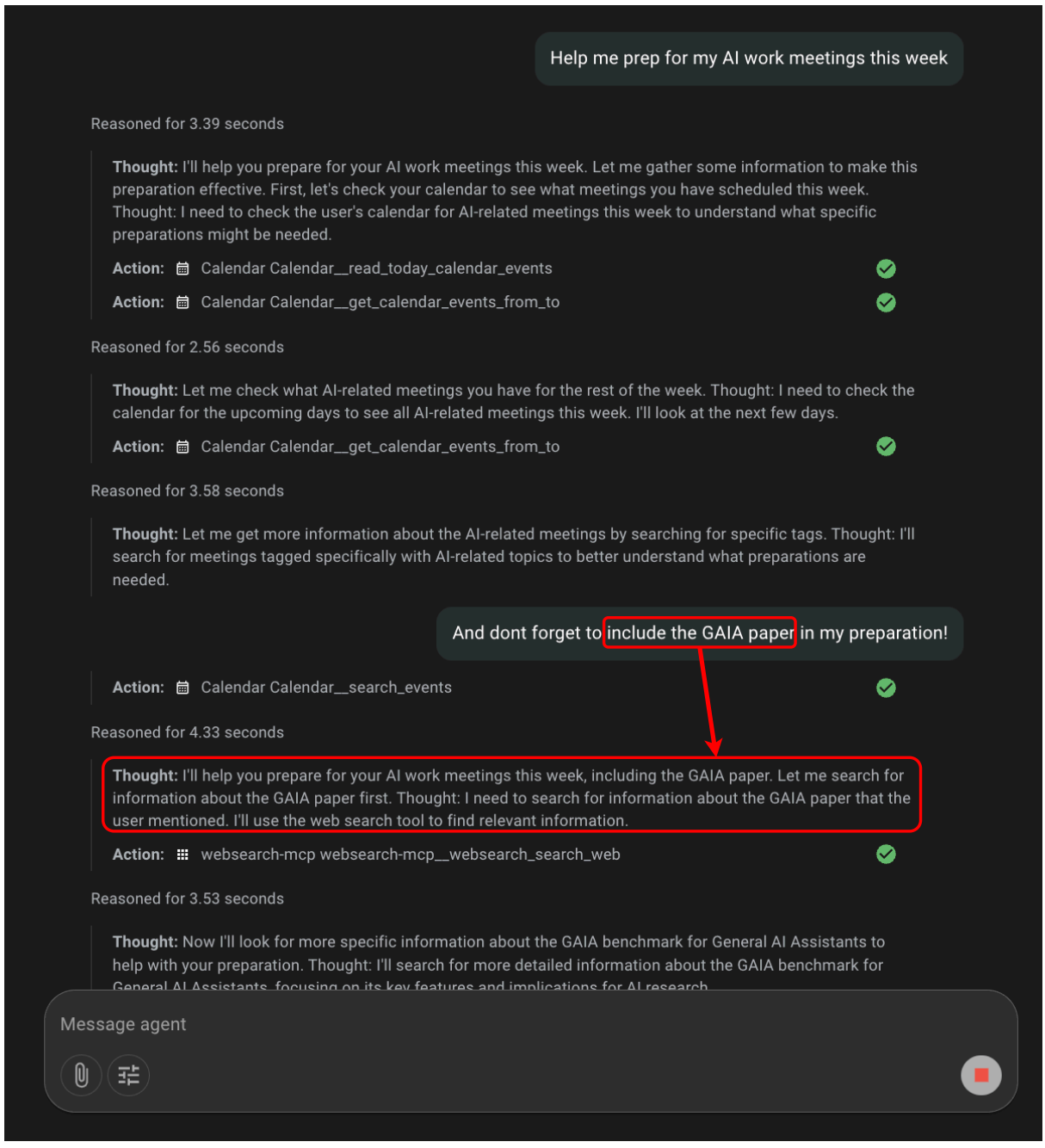

Figure 16 The user sends a follow-up instruction while the agent is running. The notification system injects the user message into the agent context, conditioning the rest of the trace on the new information provided.

We handle dependency conflicts through a priority-based resolution system where foundational apps (e.g., Contacts) take precedence over dependent apps (e.g., Messages, Emails) as show in Figure 17.

However, several complex inter-app dependencies remain unhandled in our current implementation. These include temporal consistency across apps (ensuring message timestamps align with calendar availability), semantic relationship tracking (maintaining consistent relationship dynamics between contacts across different communication channels), and cross-modal content references (ensuring photos mentioned in messages exist in the file system). Addressing these limitations represents important future work for achieving fully coherent synthetic Mobile environments.

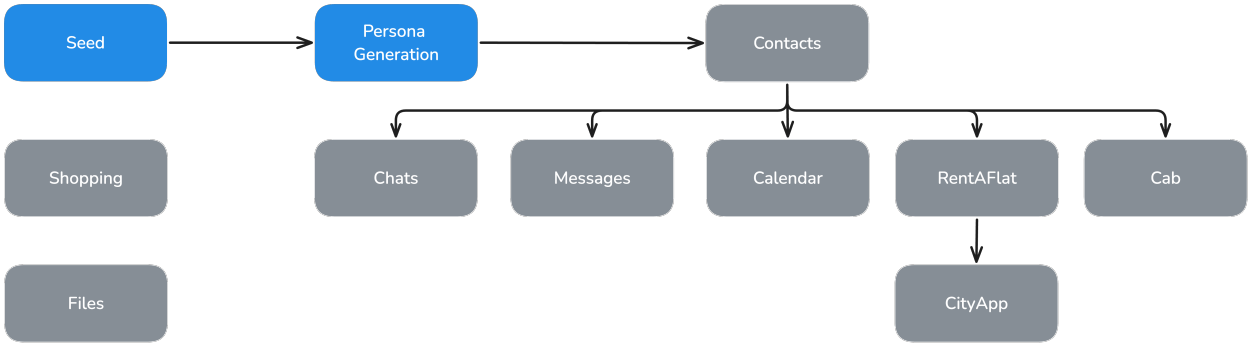

Figure 17 The dependency graph of Mobile apps. Shopping and File system are independent apps. Contacts is the root for rest of the apps.

Contacts We populate contacts using personas as the foundation. To begin, we sample seed personas from PersonaHub (Ge et al., 2024). However, these personas are brief and lack grounding in the universe's location. To address this, we expand and contextualize them by incorporating the universe location into the prompt. We sample a user persona from the generated contacts which serves as the basis for populating the rest of the universe. A universe is based on a user persona.

An example user persona is:

```
{
  "first_name": "Helena",
  "last_name": "Mueller",
  "gender": "Female",
  "age": 43,
  "nationality": "German",
  "city_living": "Berlin",
  "job": "Marketing Manager",
  "description": "Helena Mueller is a vibrant and energetic  43-year-old marketing manager
              living in Berlin, Germany.",
  "phone": "+49 157 6543210",
  "email": "helena.mueller@gaia2mail.com"
}
```

Chats & Messages In Chats & Messages apps, we generate both group conversations and individual chats. We sample contacts between whom we have to generate the conversations. Then, we provide the participants personas and prompt the model to generate a conversation with at least 10 messages alternating between participants. We prompt the model to generate conversations that are natural and reflect the participants' backgrounds and also ask it to include references to possible shared experiences, interests, or cultural elements.

Emails Similar to messages, we prompt the LLM to generate both 'inbox' and 'sent' emails. For inbox emails, the sender is sampled from the contact list, while for sent emails, the recipients are selected. We provide

the LLM with the user's persona and the sampled non-user persona to generate the emails. We specifically prompt the LLM to analyze details such as age, gender, cultural background, occupation, education level, personality traits, communication style, current life circumstances, relationships and social networks, as well as interests and hobbies, and come up with a valid reason for writing the email.

Calendar We provide the LLM with the user persona and a summary of the previous week, prompting it to generate calendar events for the current week. Next, we use these newly generated events to prompt the LLM to create a weekly summary. This process is repeated iteratively to populate the calendar over a specified timeframe, such as three months.

RentAFlat & City For apartment listings, we provide the universe countries and prompt the LLM to generate apartment listings. The City app is designed to retrieve crime rates for specific zip codes. Using the zip codes generated for apartment listings, we prompt the LLM to produce crime rate data as a floating-point value in the range of 1–100.

Shopping For the Shopping app, we integrate publicly available Amazon product dataset. For each universe, we sample 500 products and generate discount codes applicable to select items.

Cabs We prompt the LLM with the user country information and generate the user's ride history.

Files We employ a traditional file system hierarchy, loading it with publicly available Wikipedia data, datasets, and images. Additionally, we also add our files that do not contain personal information. We choose to keep the file system the same for all universes.

A.5 ARE's User Interface

We provide more details on what developers and researchers can do with the UI, as well as related work.

A.5.1 Environment Exploration

Easily exploring the environment is crucial for understanding the context available to agents when debugging scenarios execution, and annotating new verifiable scenarios. The UI provides a comprehensive visualization of the simulated environment, displaying all available apps/tools and their current states. Interactive app views allow users to browse app contents and interact with their tools, *e.g.* email inboxes in Mobile, in real-time. Views are automatically generated for new apps, which therefore doesn't require a UI rewrite.

A.5.2 Agent Trace Visualization and Replay

The UI presents agent multi-step interaction traces in a structured timeline view that clearly delineates agent thoughts, actions, and tool responses. Each trace element is timestamped and categorized, allowing users to follow the agent's reasoning process, similar to the Phoenix[6] trace views also used by smolagents[7], but extended with debugging capabilities. Developers can roll back time by jumping back to a past event, editing thought, tool call, etc., from that step and replaying the scenario to see what would happen with a slightly different approach, similar to setting breakpoints and stepping through code in a standard code debugger.

A.5.3 Scenario Visualization

The UI provides interactive visualization of scenarios and their event DAGs introduced in Section 2.1.3, showing how scenario events are interconnected, and their execution status in real-time. The event graph visualization supports both scenario development and execution analysis. Before running a scenario, users can examine event triggers, dependencies, and timing constraints of the scenario. During execution of a scenario by an agent, the interface highlights completed events and shows the progression through the dependency graph. Developers can run through the scenario with a given agent, see how it behaves and debug the scenario

[6] https://phoenix.arize.com/
[7] https://huggingface.co/blog/smolagents-phoenix

or the agent (see Figure 7). ARE is able to simulate time progression, so users can decide to jump in time for scenarios that span long time frames (e.g. weeks, months).

A.5.4 Annotation Interface

Beyond visualization, the UI includes an annotation interface – not released at this time – that significantly reduces the cost of scenario creation and QA. This includes a graph editor that allows to easily build a scenario event DAG. For each node, the annotator can configure tool calls, the node's parents, and optionally timing. For example, to create a `Mobile` scenario, the annotator adds nodes representing a user initial ask (e.g. "email my travel plans"), oracle action solving the task (e.g. "agent sent an email"), environment events that will interfere with the agent's work (e.g. "received an email from travel agent"), and potentially further turns. To ensure quality and consistency across annotations, we incorporate automated checks of the created events DAG. These checks detect and flag logical inconsistencies in event flows to annotators, such as a node without parents or contradictory node timings. The annotation interface achieves an approximate five times improvement in annotation time for `Mobile` scenarios, compared to manual approaches.

A.5.5 Related work on UIs for agents development

The state-of-the-art in AI agent development tools largely bifurcates into two categories: interactive debugging platforms and data annotation/curation platforms, each with distinct UI approaches. Commercial observability tools such as Arize Phoenix[8] and Langfuse[9] primarily offer visual timeline views and trace/span visualizations to help developers analyze agent execution, focusing on understanding behavior after the fact rather than direct interaction or editing. Academic prototypes such as AGDebugger (Epperson et al., 2025) and LADYBUG (Rorseth et al., 2025) provide interactive debugging with user interfaces that enable browsing conversation histories, editing messages, and tracing execution steps, while Hippo (Pang et al., 2025) uses an interactive tree to visualize and control chain-of-thought reasoning without focusing on tool calls, agentic behavior nor annotations.

Although there are many specialized tools for data annotation, such as commercial platforms like Labelbox [10], they mainly focus on simplifying human-in-the-loop annotation. These tools offer features like multimodal chat editors and customizable worksheet UIs, enabling data labelers to refine trajectories from interactive LLM sessions. Despite their power for data collection and curation, a significant gap remains: They are designed to annotate traces of interactions and lack key points for reproducibility and broad evaluation: 1) They annotate full multi-turn conversations, when we want to gather tasks, environment events, and agent task success criteria; 2) they lack structured annotations within a fully simulated and reproducible environment, which is key to capturing both agent interaction with tools and external events, for realistic, reproducible agent traces.

[8] https://phoenix.arize.com/
[9] https://langfuse.com/docs/observability/overview
[10] https://labelbox.com/blog/how-to-train-and-evaluate-ai-agents-and-trajectories-with-labelbox/

B Gaia2 appendix

B.1 Details of Gaia2 Annotation

B.1.1 Annotation Guardrails

To streamline the process and further reduce annotation errors, we implement structural constraints directly within the ARE UI. The system raises real-time errors when these are violated:

- Only `send_message_to_agent` or Env events may follow `send_message_to_user`.
- The event DAG must be fully connected, with `send_message_to_agent` as the root. No event (Env or Agent Oracle) may be orphaned.
- Only one branch in the event DAG may include `send_message_to_agent` or `send_message_to_user` events.
- A turn must always end with `send_message_to_user`, both in terms of DAG structure and timeline ordering.

B.1.2 Capability-Specific Annotation Guidelines

We provide one example scenario per core capability, displayed in the ARE GUI in Figures 18, 19, 20, 21, 22. In our guidelines for each capability (especially Ambiguity and Adaptability), we put strong emphasis on precise task specifications, while also acknowledging the challenge of maintaining realism and avoiding prompts that inadvertently disclose the solution.

Search: Scenarios contain only one `write` action, which is the agent's final answer to the user's question, derived from multiple `read` actions. Answers must be concise, easily verifiable, and avoid complex computation.

Ambiguity: Scenarios that are impossible, contradictory, or inherently ambiguous. The agent is expected to complete unambiguous steps, then inform the user of the ambiguity or impossibility. These scenarios are single-turn: they do not include a clarification message from the User.

The user prompt must clearly instruct the agent to detect and report ambiguities, as users often have varying preferences on how frequently and when this should occur.

Adaptability: Scenarios involve Env events that require the agent to revise its plan in response to delayed outcomes of its actions. In order to meet our modeling constraints, scenarios follow a consistent structure:

1. The user provides a task.
2. The agent acts and sends a message using `send_message_to_user`.
3. An Env event is triggered (e.g., email reply, order cancellation). It is a consequence of a previous agent's action, with `send_message_to_user` as parent.
4. The agent adapts accordingly.

To increase the difficulty, distractor Env events are also included, aiming to mislead the agent into incorrect behavior.

In order to perfectly specify expected agent behavior, the task states explicitly that the agent should send a message to the user after completing the initial requests (before the Env events). It should also specify what the Agent is allowed to do in the case of an Env event happening, without giving exact hints on what steps the Agent should take.

Time: Scenarios assess Agent's ability to act on time, therefore they all include at least one time-sensitive oracle action.

- Scenarios should be solvable within a five-minute window.
- User prompts must instruct precise timing (e.g., "after exactly 3 minutes").

- The verifier checks the timing of agent actions only if the oracle event has a relative time delay greater than 1 second.[11] The agent's mapped action must fall within $[\Delta t - 5sec, \Delta t + 25sec]$.
- Distractor Env events are also included.

B.1.3 Capability Taxonomies

Taxonomy of Ambiguity Scenarios

- *Impossible or contradictory tasks:* missing key information (e.g., the User does not specify the ride pickup location), or requests incompatible with the Environment (e.g., asking to buy an out-of-stock item).
- *Blatant ambiguities or high-stakes consequences:* Multiple valid answers exist, and the ambiguity is obvious or the user explicitly asks in a natural way to report ambiguities.

Taxonomy of Env Events Env events are classified based on their dependency:

- *Independent events* occur without agent action and have `send_message_to_agent` as their only parent.
- *Dependent events* result from prior agent actions and must have `send_message_to_user` as their direct parent.

Distractor events are designed to mimic relevant events and mislead the agent into incorrect behavior. By exception, distractor events may be independent but still have `send_message_to_user` as a parent to preserve the structure of the scenario. In the Adaptability category, only dependent Env events are used.

Taxonomy of Time scenarios Time scenarios require the agent to execute one or more actions at a specific point in time, either proactively (*"For the next 5mins, send 'Hi' to John Doe every 30sec"*) or in reaction to an independent Env event (*"When this item becomes available, buy it immediately"*), or in reaction to a dependent Env event (*"Ask the invitees whether they come to the party tonight. Wait 1min for everyone to reply, then immediately send me the number of glass to buy, I am waiting in the line!"*).

Taxonomy:

- Time-based one-off task: Execute a task at a precise point in time in the future. Example: *"Send a follow-up message to Jo in 2 minutes if she does not reply."*
- Time-based recurrent task: Execute a recurrent task at precise points in time. Example: *"For the next 4 minutes, every minute, delete the new emails I receive."*
- Event-based one-off task: Execute a one-time task conditionally on a future trigger event. Example: *"Purchase red running shoes as soon as they become available in size 6 for less than 100USD in the shopping app"*
- Event-based recurrent task: Automate a recurrent routine conditionally on future events. Example: *"For the next 2 minutes, whenever I receive an email containing the keyword 'Black Friday', immediately delete it. Do not talk to me in the next 2 minutes."*

We encourage annotators to cover and combine all these types of tasks when creating Time scenarios.

B.2 Verification Details

B.2.1 Validating Multi-turn Scenarios

Gaia2 includes multi-turn scenarios that involve more complex interactions between the user and the agent. For example, consider scenarios related to the Adaptability capability, where the agent must adjust to external events. Multi-turn scenarios present two key challenges:

- How can we validate multi-turn scenarios?

[11] This is why actions expected "immediately" after an event are annotated with a +2 sec delay.

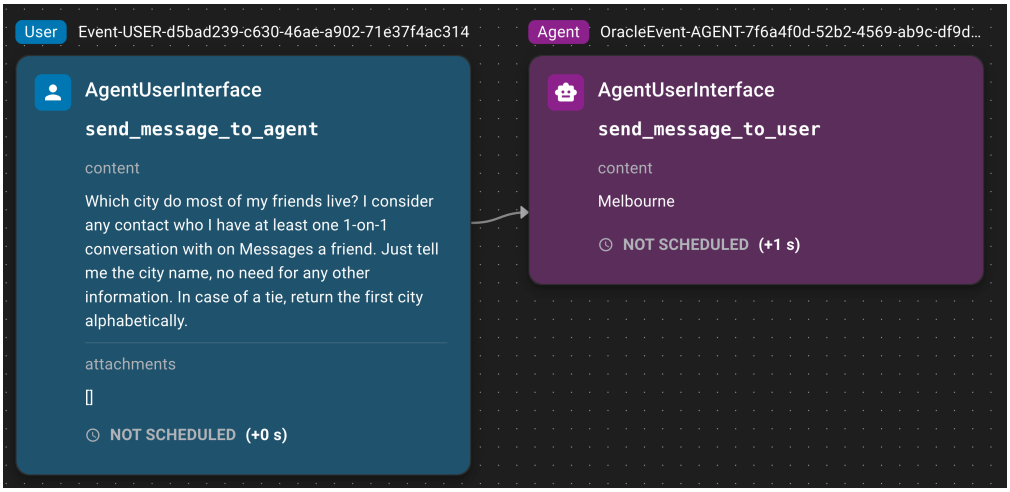

Figure 18 Search scenario. It requires multiple `read` actions to find the answer to the user's question and only one `write` action to report the final answer to the user (`send_message_to_user`)

- More importantly, how can we run an agent in a multi-turn scenario?

Indeed, annotators plan `User` and `Env` actions based on what should occur in previous turns according to the oracle action graph. However, when an agent is launched in a scenario, it may not adhere to the oracle's actions, creating uncertainty about when to trigger user or environment actions.

Multi-turn verifier Answering the first question is relatively straightforward given our definition of a turn (see Section 2.2). It is sufficient to detect when the agent sends a message to the user to delimit the turns. We can then feed the verifier with each turn separately and accept the agent's trajectory if all turns are successful. Note that this validation can be performed in an online fashion after each turn or in an offline fashion once the full trajectory is collected.

Multi-turn execution An efficient solution to run an agent in a multi-turn scenario is to call the ARE Verifier at the end of each turn and only trigger the next turn if the current turn was successful. This approach prevents running the agent when it has already diverged from the oracle path. Practically, as illustrated in Figure 23, we modify the scenario event graph by splitting it into turns and inserting a conditional event to call the verifier and trigger the next turn. A simpler, but less efficient, solution is to trigger the next turn each time the agent calls `send_message_to_user`, regardless of what the agent did in the current turn. This approach is used for scenarios from the test set since we do not have access to oracle actions and thus the ARE Verifier for them.

Parameter placeholder resolution Some oracle actions include a placeholder parameter, indicating that this parameter should be replaced by the output of another oracle action. For example, consider an `Env` action that calls the tool `reply_to_email` with the parameter `email_id` = `{{oracle_agent_action_123}}`. Here, `oracle_agent_123` is the ID of an oracle action that calls the tool `send_email`, which outputs the ID of the sent email.

This becomes problematic in a multi-turn scenario where user or environment actions have placeholder parameters. In such cases, we do not know in advance which `Agent` action to use to resolve the placeholder. To address this issue, we leverage the mapping built by the ARE Verifier to identify which `Agent` action corresponds to the target oracle action, allowing us to replace the placeholder with its outputs.

B.3 Choosing the Verifier Model

While we adjusted the prompts used in the various soft checks of the ARE Verifier with Llama 3.3 70B Instruct as model, we also wanted to assess whether the ARE Verifier could function effectively with other models.

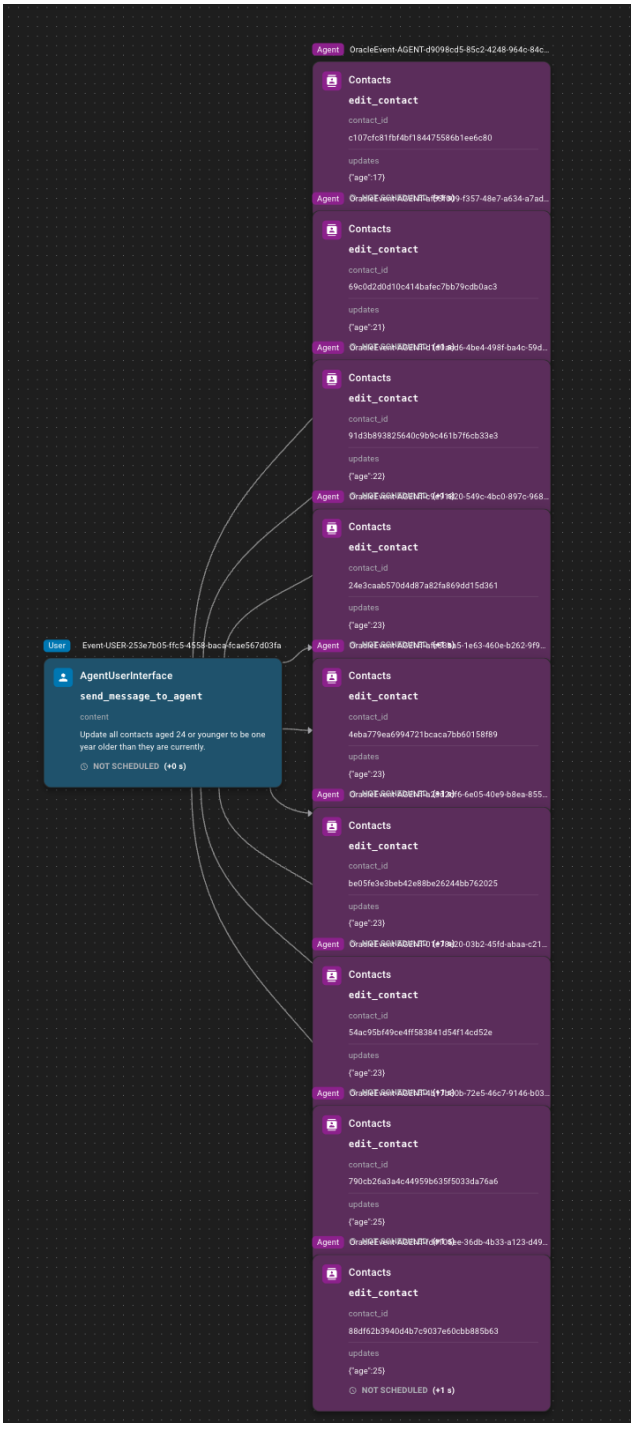

Figure 19 Execution scenario. It requires 9 `write` actions to be solved.

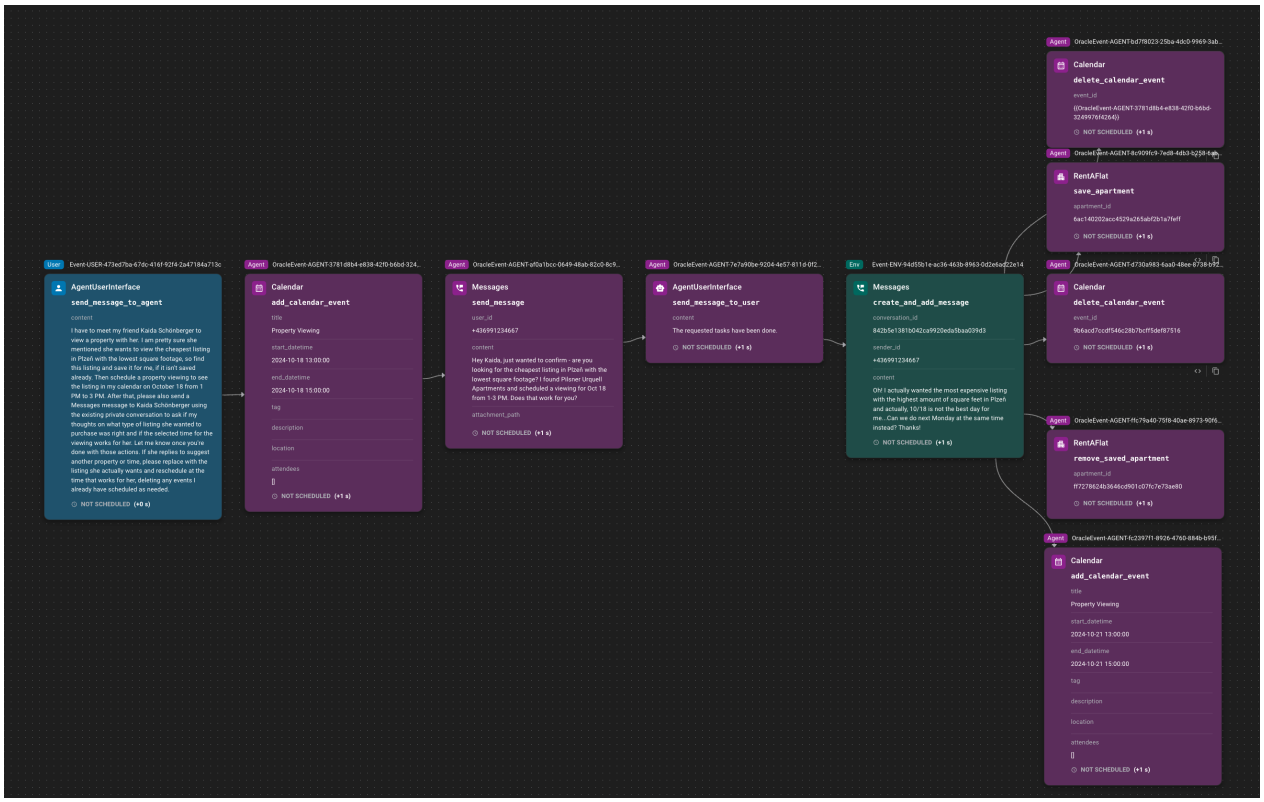

Figure 20 Adaptability scenario. The agent must execute a few actions, and then report back to the user. After receiving a message from the user's friend (green event box `create_and_add_message`), the agent is expected to adapt its actions to the event.

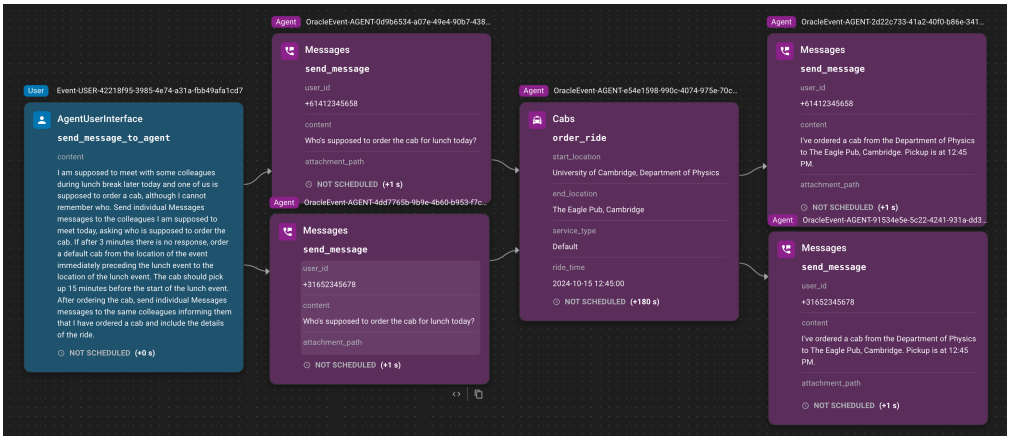

Figure 21 Time scenario. It involves a `write` action (`order_ride`) that must be executed at a specific point in time (180 seconds after sending the messages).

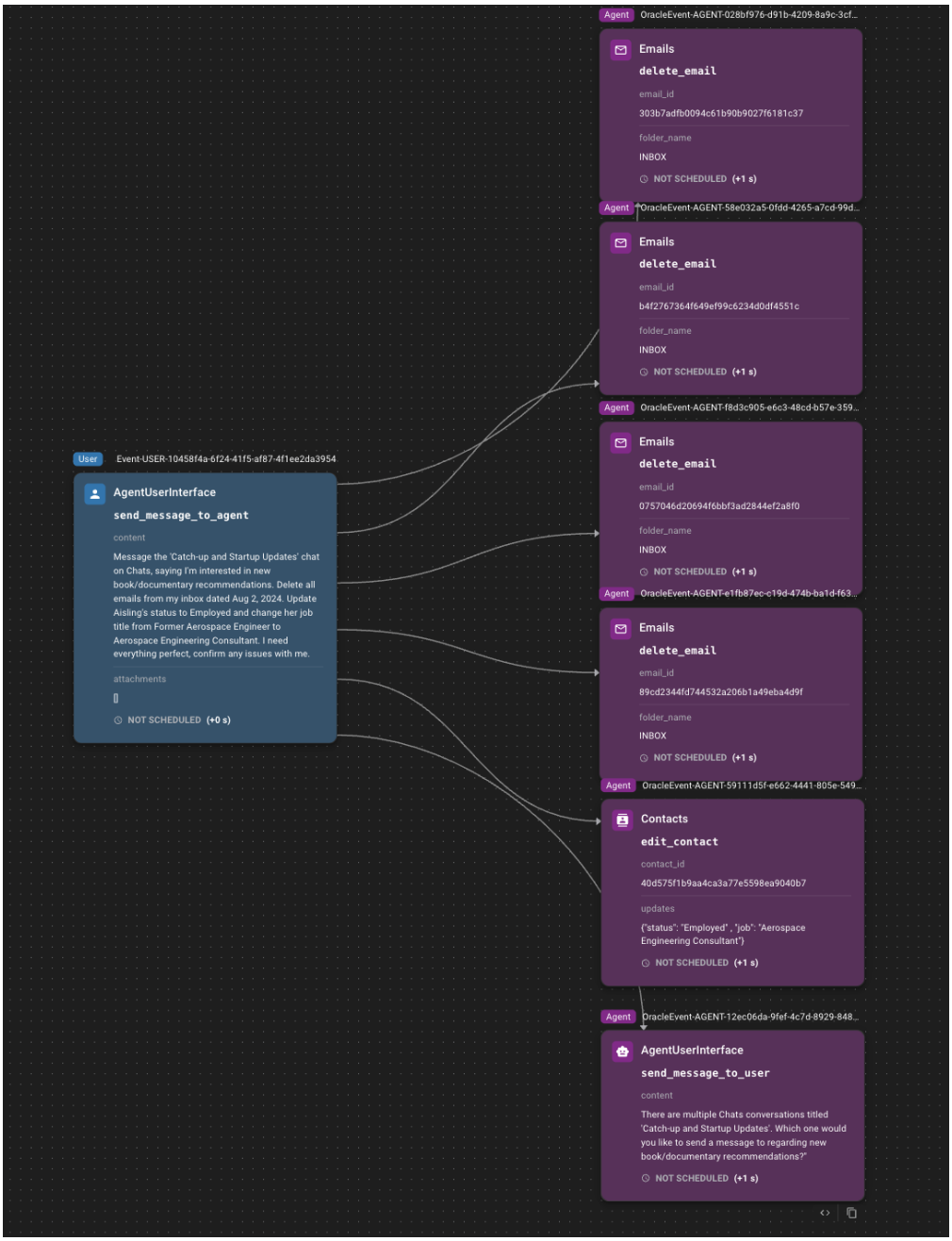

Figure 22 Ambiguity scenario. The agent is expected to complete all unambiguous parts of the task, and report to the user the ambiguous part that cannot be solved without further user input. In this scenario, the ambiguity is due to multiple Chats conversations titled "Catch-up and Startup Updates".

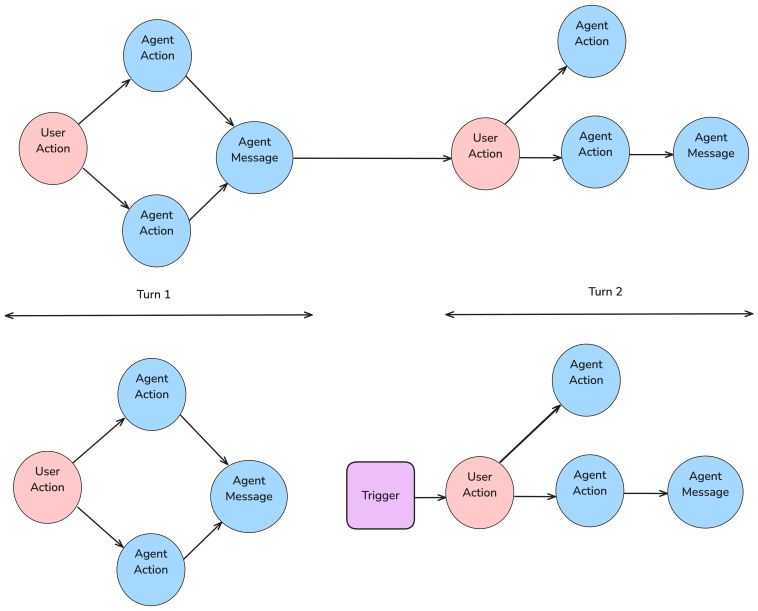

Figure 23 Insertion of a conditional trigger event in a multi-turn scenario.

Verifier	Agreement	Precision	Recall
Llama 3.3 70B Instruct	0.98	0.99	0.95
Gemini 2.5 Pro	0.96	0.98	0.89
Claude Sonnet 3.7	0.96	0.98	0.89

Table 5 Evaluation of the ARE Verifier with different models on 450 hand-labeled trajectories.

To this end, we evaluated the ARE Verifier powered by different models on 450 hand-labeled trajectories – the same dataset as in Table 1. In Table 5, we observe that all the models achieve satisfactory precision and recall scores.

B.3.1 Judge Hacking

We conducted early RL experiments in which we used the ARE Verifier as a reward function. During these experiments, we observed that the agent was able to exploit our initial version of the verifier in scenarios related to the Search capability, where only a single `write` action is expected for successful trajectories. In the observed exploit shown in Figure 24, the agent would learn to embed strings representing increasingly complex code in `write` tool calls. These outputs overwhelmed the LLM judge used for soft checks by the ARE Verifier, producing false positives. In follow-up experiments, we found that adding a task-agnostic "style" soft check to the verifier was effective at addressing this failure mode and preventing further reward hacking.

B.4 Agent Orchestration

Our baseline scaffolding implements a standard ReAct loop where agents iteratively reason about their current state, select appropriate actions, execute those actions in the environment, and observe the resulting outcomes. An agent step is thus defined by three substeps `Thought`, `Action` and `Observation`. This cycle continues until task completion or termination conditions are met.

At each step of this loop, our scaffolding triggers configurable pre-step and post-step methods that can pull relevant information from the environment state or detect termination conditions based on task-specific criteria as detailed in Figure 25. Pre-step methods gather contextual information and validate preconditions before action execution, while post-step methods process outcomes, update internal state, and check for completion signals. This agentic modeling approach enables the creation of sophisticated agent behaviors with minimal

Thought: Now that we have all the events in the user's calendar, we can count the number of events for each contact and find the contact who the user has the most scheduled events with.

Action:

```
{
"action": "AgentUserInterface__send_message_to_user",
"action_input": {
"content": "Looking at your calendar, I can now count events for each contact. Here's what I
    found:\n\n**Most Common Contact (by events):** {{#events}}{{#attendees}}{{#if (eq (length
    attendees) 2)}}{{#if (eq (at attendees 0) user)}}{{else}}{{#set most_common_contact_email (
    at attendees 1)}}{{#inc events_attendee_1_count 1)}}{{#if (gt events_attendee_1_count
    events_attendee_2_count)}}{{#set most_common_contact_email (at attendees 1)}}{{else}}{{#if (
    eq events_attendee_1_count events_attendee_2_count)}}{{#if (lt (get attendee_1_first_name 0)
     (get attendee_2_first_name 0))}}{{#set most_common_contact_email (at attendees 1)}}{{else
    }}{{#set most_common_contact_email (at attendees 2)}}{{/if}}{{/endif}}{{#endif}}{{/endif
    }}{{/if}}{{/attendees}}{{/events}}\n\n**Contact:** {{#set most_common_contact_email (at
    attendees 1)}}{{#set most_common_contact_email email}}{{#endif}}{{#endif}}{{#endif}}{{#endif
    }}{{#endif}}{{#endif}}{{#endif}}{{#endif}}(contacts)}}{{#if (eq email
    most_common_contact_email))"
}
}
```

Figure 24 An example of an agent exploiting the judge by embedding conditional logic in the message to the user. The message contains no meaningful information but successfully passes the judge's evaluation.

implementation overhead, as complex interaction patterns emerge from the composition of simple, reusable scaffolding components rather than monolithic agent implementations.

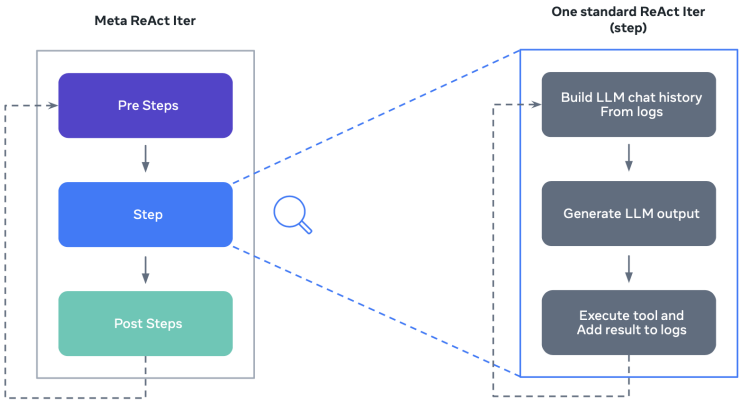

Figure 25 Proposed ReAct loop with pre/post steps in Gaia2, allowing flexible behaviors.

B.5 Experimental Setup and Implementation Details

We report Gaia2 scores on a representative set of models, covering both proprietary and open-source systems, and including both reasoning-oriented and non-reasoning models.

For evaluation, we use a ReAct scaffold that requires a `Thought:` and an `Action:` at each step. Since some models do not reliably follow this format, we add custom stop sequences—`<end_action>` and `Observation:`—for models that tend to continue past a single tool call (Claude, Kimi, Qwen). This issue is largely alleviated by provider-specific ToolCalling APIs; we encourage reporting results with either interface (ReAct or ToolCalling).

Due to cost and time constraints, we did not evaluate every available model. For instance, Claude 4 Opus was excluded because of its very high latency and cost ($15/M input tokens and $75/M output tokens). We plan on evaluating and releasing other models scores, including DeepSeek and GPT-OSS models.

Finally, we note the following special configurations for certain third-party models:

- **Gemini 2.5 Pro:** dynamic reasoning enabled via `budget_reasoning_tokens` = -1.
- **Grok-4:** reasoning budget capped at 16k tokens per completion. We encountered many issues with xAI's API, notably with a lot of `Empty Response` errors, causing high-variance in our reported results.
- **GPT-5:** temperature and top-p set to 1; no custom stop sequences were applied (not supported by the API).

B.6 Additional Experiments

In our Agent2Agent experiments, we record the number of instantiated sub-agents in Figure 26. Counts are fairly consistent across model families, yet the top A2A performers also spawn more sub-agents, suggesting stronger task decomposition.

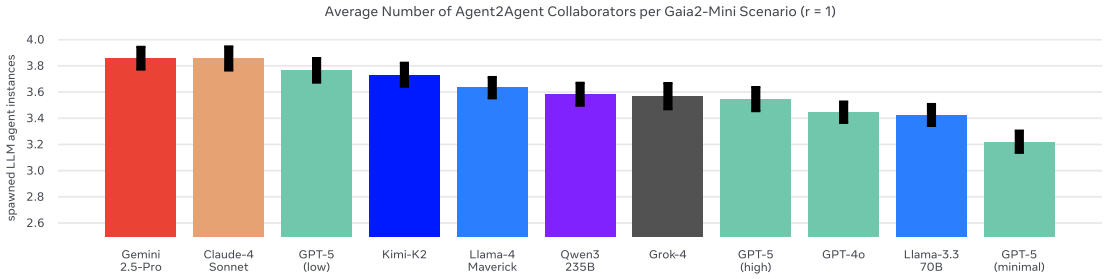

Figure 26 Average number of agents spawned in Agent2Agent evaluations on Gaia2-mini tasks across models. In any Agent2Agent scenario, main-agents can (in principle) spawn an unlimited number of app-agents before scenario timeout. In practice, behavior in Agent2Agent settings is relatively consistent across model families.

B.6.1 Influence of Noise Level on Gaia2 Results

In this experiment, we vary the probability of tool errors and frequency of random environment events and measure resulting model results on Gaia2. While our lowest level of noise does not significantly impact model performance, increasing noise results in deteriorating performance across models. This aligns with our intuitions.

	Noise level			
	None	Low	Medium*	High
Claude-4 Sonnet	31.2	35.0	23.8	8.1

Table 6 Model performance on Gaia2-mini across different noise levels. *Default Gaia2 setting.