General Agent Evaluation

Elron Bandel [1] **Asaf Yehudai** [1] **Lilach Eden** [1] **Yehoshua Sagron** [1] **Yotam Perlitz** [1] **Elad Venezian** [1]
Natalia Razinkov [1] **Natan Ergas** [1] **Shlomit Shachor Ifergan** [1] **Segev Shlomov** [1] **Michal Jacovi** [1]
Leshem Choshen [1 2] **Liat Ein-Dor** [1] **Yoav Katz** [1] **Michal Shmueli-Scheuer** [1]

Abstract

General-purpose agents perform tasks in unfamiliar environments without domain-specific manual customization. Yet no study has systematically measured how agent architecture shapes performance across heterogeneous protocols and diverse unfamiliar environments. This is the first systematic study, comparing tool-calling, MCP, code-generation, and CLI agents on the same benchmarks with the same models. Two gaps blocked it: existing harnesses fix the protocol or require per-benchmark wiring, and benchmarks expect human-authored prompts and integration glue. To enable the study, we contribute a unifying protocol, an evaluation harness that surfaces any benchmark to any agent and backbone model, and the first Open General Agent Leaderboard of agent configurations—a factorial over 5 agent architectures $\times$ 5 backbone LLMs $\times$ 6 benchmarks spanning software engineering, customer service, deep research, and personal assistance. We find: (i) general agents adapt to every tested domain without per-domain tuning; (ii) architecture choice swings results up to 12pp within a single model, yet backbone model choice dominates overall; (iii) on 4 of 6 benchmarks, top general agents match heavily-customized specialists; (iv) the open-weight models tested exhibit "generality sinks"—collapse on specific architectures or benchmarks—absent from frontier closed-source; (v) failure analysis reveals architecture-distinctive error signatures invisible to aggregate scoring. Code, harness, leaderboard, and traces at `www.exgentic.ai`.

[1]IBM Research [2]MIT. Correspondence to: Elron Bandel <elron.bandel@ibm.com>.

Published at the Second Workshop on Agents in the Wild: Safety, Security, and Beyond (AIWILD) at ICML 2026. Copyright 2026 by the author(s).

1. Introduction

Recent AI agents tackle software engineering, navigate web interfaces, and handle multi-application workflows with growing proficiency (Zhang et al., 2024; Deng et al., 2023). However, current progress largely relies on domain specialization and manual tuning, whereas heterogeneous real-world settings demand general-purpose agents capable of scalable deployment without such manual customization (c.f., Marreed et al., 2025; Bandel et al., 2026a).

Despite their importance, current evaluation practices cannot adequately assess general-purpose agent capabilities. Existing agentic benchmarks like SWE-Bench Verified (Jimenez et al., 2023) and τ^2-Bench (Yao et al., 2024) provide valuable assessments of domain-specific agents. Yet, they impose two constraints preventing general-agent evaluation: they use bespoke communication protocols (Lacoste et al., 2026), and they implicitly assume agents have prior knowledge of benchmark-specific goals and environment semantics (Bandel et al., 2026b). Recent consolidation efforts like BrowserGym (Chezelles et al., 2025) and Harbor (Shaw, 2025) have integrated multiple benchmarks within single domains by exposing the agent to current goals and environment semantics. While a step forward, these frameworks still enforce a single integration interface (web-based for BrowserGym, CLI-based for Harbor), preventing agents from using their native integration mechanisms and effectively evaluating a diminished version of the agent (Yehudai et al., 2025).

Recent works call for evaluating general agents as a research target (Bandel et al., 2026a), but no such evaluation exists today. This paper addresses this gap: no prior study evaluates the same unmodified agent across multiple benchmarks, so the contribution of the agent itself cannot be isolated. Two technical problems blocked such a study: *(i) A framework gap:* no evaluation harness allows any agent to be plugged into a new benchmark unchanged (e.g., Harbor (Shaw, 2025) requires agents to integrate with its terminal API); even the most mature ones (e.g., Inspect (AI Security Institute, UK, 2024)) require manual per-benchmark wiring of tools, sandbox, and solver. *(ii) A benchmark gap:* contemporary benchmarks encode task information, affor-

Table 1. Positioning of this work relative to prior agent-evaluation studies and frameworks along five axes. ✓ indicates the property holds; ✗ indicates it does not; ✓× indicates it holds within a constrained class. *Multi-Protocol Benchmark*: benchmarks are supported in their native form across heterogeneous protocols, without forcing a fixed transport. *Multi-Protocol Agent*: agents are supported in their native form across heterogeneous protocols. *Agent Plug-and-Play*: an agent integrates without per-benchmark adaptation. *Systematic Factorial Study*: The work perform a comparative study that decomposes performance across the full agent-architecture × backbone-model × benchmark factorial. *General-Purpose Tasks*: evaluated tasks span domains rather than a single application class. Per-cell justifications are provided in App. A.

System	Multi-Protocol Benchmark	Multi-Protocol Agent	Agent Plug-and-Play	General-Purpose Tasks	Systematic Factorial Study
agent-eval (Bragg et al., 2025)	✓	✓	✗	✗	✓×
HAL (Kapoor et al., 2025)	✓	✓	✗	✓	✓×
Inspect (AI Security Institute, UK, 2024)	✓	✓	✗	✓	✗
BrowserGym (Chezelles et al., 2025)	✗	✗	✓	✓×	✓×
Harbor (Shaw, 2025)	✗	✗	✓	✓×	✗
AgentBeats	✗	✗	✓	✓	✗
CUBE (Lacoste et al., 2026)	✗	✗	✓	✗	✗
This work	✓	✓	✓	✓	✓

dances, and interaction, assuming manual integration, and do not expose the means needed for a general-agent interface (Table 1 contrasts existing systems along these axes).

This paper closes the two technical gaps above and presents the first systematic study of diverse general agents across diverse environments (Table 1 positions this work relative to prior evaluation systems). In this work, a *general agent* performs tasks in unfamiliar environments without domain-specific customization; we test each agent unmodified on benchmarks it was not customized for and measure its cross-benchmark success. We release three artifacts: (1) the *Unified Protocol*, a benchmark-agent mediation protocol that bridges agent interfaces (CLI, tool-calling APIs, MCP) and benchmarks through a canonical task/context/actions representation; (2) *Exgentic*, an evaluation harness implementing the Unified Protocol and surfacing benchmarks through an interface accessible to any general-purpose agent with any backbone model; (3) the first public Open General Agent Leaderboard, at a total evaluation cost of $20K (Table 2).

Our analysis of the Open General Agent Leaderboard yields five findings. **(1) General agents adapt without manual tuning:** we run each unmodified across software engineering, customer service, technical support, deep research, and personal assistance benchmarks, with no per-domain engineering, producing non-trivial performance on every domain. **(2) Model choice dominates aggregate variance, but architecture is conditionally decisive:** model choice explains 27.8% of variance versus 0.5% for architecture, a 58× aggregate gap across closed-source configurations, yet within a single model architecture choice swings results up to 12pp (Fig. 1). **(3) Generalists are competitive with heavily-customized specialists:** on 4 of 6 tested benchmarks, the best general-agent configuration is statistically indistinguishable from the top published domain-specific score; BrowseComp+ and τ^2-Bench-Telecom remain specialist-led (Tab. 3). **(4) The two open-weight models tested (DeepSeek-V3.2, Kimi-K2.5) exhibit "generality sinks" absent from frontier closed-source:** on τ^2-Bench the same backbone swings 0.83 to 0.00 across architectures (architecture sink, traceable to first-turn protocol violations), and on AppWorld every architecture collapses (benchmark sink). **(5) Failure modes discriminate architectures that aggregate scoring cannot:** although architecture explains only 0.5% of success-rate variance, agents fail in characteristic ways: Claude Code and OpenAI Solo tend to stop too early, while ReAct and ReAct Short tend to skip evidence gathering (§4.10).

Ultimately, advancing general-purpose agents requires a collective effort. We hope the Open General Agent Leaderboard serves as a catalyst for approaches that transcend individual tasks and invite the research community to expand this ecosystem by contributing benchmarks that challenge generalization and by designing novel evaluation protocols.

2. Unified Protocol Methodology

To enable rigorous study of general agents, this work provides a solution for evaluating diverse agents on diverse agentic benchmarks, even when they natively communicate with different protocols. Existing approaches block wide agent evaluation by either restricting to single-protocol harnesses (e.g., BrowserGym, Harbor) or requiring costly per-benchmark integration. We introduce a Unified Protocol that serves as a faithful mediation layer between agents and benchmarks.

Table 2. The first Open General Agent Leaderboard, ranking all 25 (agent architecture × backbone model) configurations on standardized benchmarks. Each cell reports binary success rate; Avg is the benchmark-weighted average; Cost is average USD per task. Per-cell Wilson 95% CIs span ±7–13pp; aggregate significance tests in App. F. Zero scores are discussed in §4.7.

#	Agent	Model	App	BC+	SWE	T-Air	T-Ret	T-Tel	Avg	Cost
1	OpenAI Solo	Claude Opus 4.5	0.68	0.61	0.81	0.74	0.85	0.84	0.73	$8.5
2	Claude Code	Claude Opus 4.5	0.66	0.53	0.74	0.66	0.83	0.76	0.67	$8.0
3	Smolagent	Claude Opus 4.5	0.70	0.61	0.65	0.72	0.78	0.58	0.66	$4.4
4	ReAct Short	Gemini 3	0.55	0.48	0.71	0.70	0.82	0.73	0.62	$0.7
5	ReAct Short	Claude Opus 4.5	0.64	0.49	0.61	0.66	0.78	0.76	0.62	$3.8
6	ReAct	Gemini 3	0.50	0.48	0.71	0.70	0.82	0.73	0.61	$0.8
7	ReAct	Claude Opus 4.5	0.61	0.49	0.61	0.66	0.78	0.76	0.61	$5.8
8	OpenAI Solo	Gemini 3	0.57	0.33	0.72	0.62	0.73	0.79	0.59	$2.8
9	Claude Code	Gemini 3	0.36	0.51	0.67	0.70	0.71	0.71	0.56	$2.5
10	Smolagent	Gemini 3	0.13	0.57	0.76	0.68	0.75	0.88	0.56	$1.8
11	ReAct Short	GPT 5.2	0.22	0.46	0.57	0.54	0.73	0.53	0.46	$0.3
12	ReAct	DeepSeek-V3.2	0.09	0.36	0.69	0.56	0.82	0.71	0.46	$0.4
13	ReAct Short	DeepSeek-V3.2	0.04	0.36	0.69	0.56	0.82	0.71	0.45	$0.2
14	ReAct Short	Kimi-K2.5	0.10	0.34	0.57	0.62	0.65	0.83	0.43	$0.4
15	ReAct	Kimi-K2.5	0.09	0.34	0.57	0.62	0.65	0.83	0.43	$0.5
16	Smolagent	Kimi-K2.5	0.11	0.33	0.58	0.56	0.72	0.71	0.42	$0.7
17	Claude Code	DeepSeek-V3.2	0.03	0.48	0.64	0.28	0.65	0.61	0.42	$0.2
18	Smolagent	DeepSeek-V3.2	0.13	0.21	0.56	0.60	0.77	0.84	0.41	$0.2
19	ReAct	GPT 5.2	0.00	0.46	0.57	0.54	0.73	0.53	0.41	$0.2
20	Claude Code	GPT 5.2	0.00	0.43	0.58	0.48	0.64	0.55	0.39	$0.4
21	OpenAI Solo	GPT 5.2	0.00	0.48	0.55	0.50	0.53	0.53	0.39	$0.2
22	Smolagent	GPT 5.2	0.07	0.26	0.53	0.60	0.68	0.71	0.38	$0.4
23	OpenAI Solo	DeepSeek-V3.2	0.06	0.30	0.74	0.20	0.19	0.18	0.32	$0.1
24	Claude Code	Kimi-K2.5	0.08	0.56	0.52	0.12	0.03	0.00	0.30	$0.6
25	OpenAI Solo	Kimi-K2.5	0.08	0.35	0.57	0.00	0.01	0.00	0.25	$0.2

The Unified Protocol serves as a "narrow waist": adding a new agent (or benchmark) only requires adhering to the Unified Protocol, not to every benchmark (or agent). Thus, it significantly reduces integration complexity, development effort and learning curve.

The Unified Protocol is not an imposed standard; we derived it by surveying existing agent and benchmark communication patterns and extracting their common structure. By construction, every semantic these protocols express is faithfully representable in the Unified Protocol, so nothing is lost when translating between them.

2.1. Agent Benchmark Unified Protocol

The protocol defines instances that are passed between the benchmark and the agent. Each instance has three fields: task, context, and actions. Here we demonstrate them with τ^2-Bench as our running example (see other benchmark examples in Appendix C).

Task (*what the agent should do*): a textual description of the task. In τ^2-Bench, it is *"You are a customer service agent that helps the user according to the policy provided below. Try to be helpful and always follow the policy."*; the first user utterance, such as *"Cancel my flight reservation AH3BDS"*, is passed to the agent separately as the first observation from the environment. **Context** (*what the agent should know*): additional information provided to the agent to accomplish the task. In τ^2-Bench, the context contains the *policy*; the agent can use it in different ways (naively append to the task, store in a dedicated memory or document store for conditional retrieval). **Actions** (*what the agent can do*): a set of environment

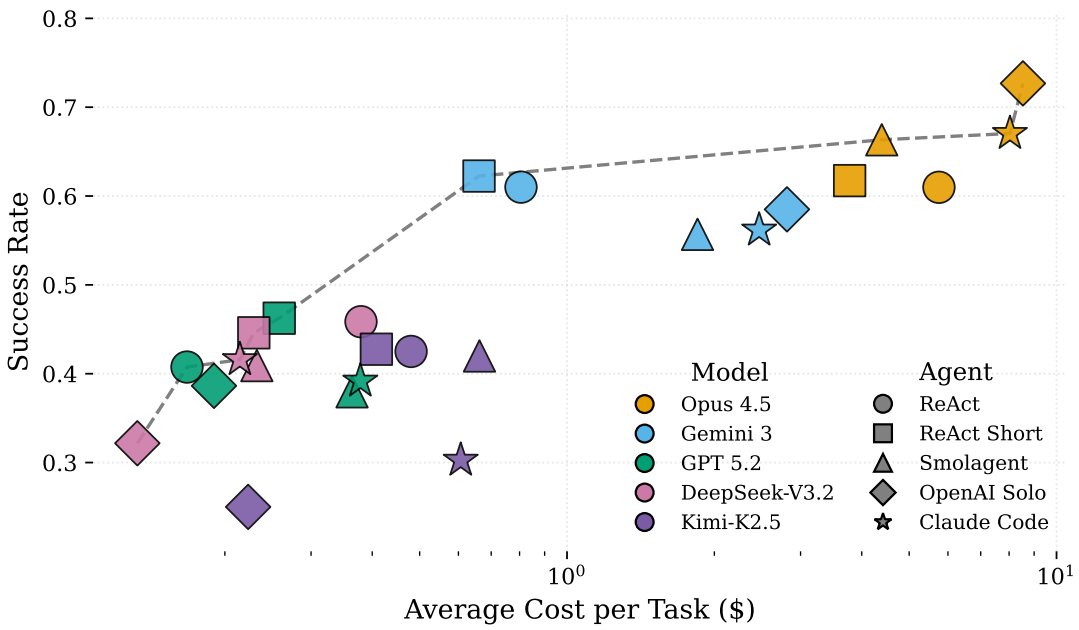

Figure 1. Cost-performance tradeoffs across all 25 agent-model configurations (cost on log scale). The Pareto frontier (dashed line) shows optimal tradeoffs: DeepSeek-V3.2- and GPT 5.2-backed configurations occupy the cost-efficient end, while Claude Opus 4.5-backed configurations (notably OpenAI Solo+Claude Opus 4.5) achieve the highest performance at 1–2 orders of magnitude higher cost. Each point aggregates $n = 550$ tasks; model-axis contrasts aggregate $n = 2{,}750$ per backbone model, giving tight CIs of $\sim \pm 1.8$pp (App. F).

actions constituting the complete operations available for performing the task. Each action specifies a typed set of parameters and may return one or more observations of arbitrary types. In τ^2-Bench (airline domain), example actions are $\text{cancel_reservation}(reservation_id)$ and $\text{search_direct_flight}(origin, destination, date)$.

Reviewing existing protocols and agents, we observed that many introduce special handling for two specific types of interactions with the environment: (1) sending a message to a user, and (2) submitting a final answer to the benchmark, signaling that the agent has completed the task. To support these common interaction patterns, the Unified Protocol allows implementers to optionally designate one action as the message action and one as the final-answer action.

The three-field task/context/actions representation is minimal by design, accommodating any agent protocol decomposable into discrete actions.

2.2. Running the Evaluation

We manually adapted six benchmarks and five agents into Exgentic, a unified evaluation harness. These benchmarks and agents use heterogeneous interaction protocols: tool-calling APIs (ReAct agent, τ^2-Bench benchmark), MCP (OpenAI Solo, Claude Code agents), Python code generation (Smolagent agent, AppWorld benchmark), bash/CLI (Claude Code agent, SWE-Bench Verified benchmark), and conversational messaging (τ^2-Bench benchmark).

For each benchmark, we derived a Unified Protocol interface from a reference agent implementation, preserving the benchmark's intended semantics. We then implemented wrapper code that adapts the original benchmark and agent implementations to this Unified Protocol interface.

The Exgentic evaluation harness executes the original agents and benchmarks as black boxes. The Exgentic orchestrator instantiates the corresponding Unified Protocol wrappers and executes 150 agent $\times$ model $\times$ benchmark configurations in isolated, reproducible sessions.

Appendix A details one complete adaptation example (Mini-SWE agent→SWE-Bench Verified) and describes the orchestrator design.

3. Experimental Setup

We evaluate 5 agent architectures across 5 LLMs (3 frontier closed-source: GPT 5.2 (OpenAI, 2025b), Claude Opus 4.5 (Anthropic, 2025), Gemini 3 Pro (Google DeepMind, 2025); 2 open-weight: DeepSeek-V3.2 (DeepSeek-AI, 2025), Kimi-K2.5 (Kimi Team, 2026)) on 6 benchmarks, with 100 tasks per benchmark (50 for τ^2-Bench Airline) and a 100-turn cap per task, following Perlitz et al. (2024), yielding 150 configurations. By *agent ar-*

chitecture we mean the full bundle shipped with the agent: scaffold, available tools, memory, schema guards, and any auxiliary components — everything the agent author commits to except the backbone model. LLM sampling uses each provider's documented defaults (temperature, top-p, reasoning mode), chosen to avoid confounding the model-versus-agent comparison with hyperparameter tuning. Full run configurations are released with the code.

3.1. Benchmarks

Appendix C provides detailed descriptions of the benchmark adaptations to the Unified Protocol.

BrowseComp+ (Chen et al., 2025) is a deep research benchmark to assess an agent's ability to handle complex information-search tasks involving iterative search planning and multi-step reasoning. While the original benchmark jointly evaluates LLMs and retrieval components, we fix the retriever to isolate agent reasoning and decision-making. We use the authors' provided retriever with either BM25 (Robertson et al., 1994) or Qwen3 Embedder-based dense retrieval (Zhang et al., 2025), and report results using the latter.

τ^2-Bench evaluates customer-service agents across retail, airline, and telecom domains via LLM-simulated users, measuring both policy-compliant task completion and violation rejection. τ^2-Bench has a bespoke Python API, where the agent receives a simulated user message and returns either a message reply or calls to one or more predefined tools. We map these into a *message* action and Exgentic actions respectively.

SWE-Bench Verified A human-validated subset of real-world software engineering tasks from popular Python repositories. Each provides a GitHub issue and repository snapshot; agents produce patches that are evaluated against hidden test suites. Following mini-swe-agent (SWE-agent Team, 2024; Yang et al., 2024) (Tab. 6), we expose a single `bash` action for repository interaction in a sandboxed environment, generating patches via `git diff` for evaluation. This ensures uniform agent interaction.

AppWorld is a benchmark for evaluating user-assistance agents on realistic day-to-day digital tasks. In the original protocol, the agent interacts with the environment by writing Python code that is executed in a dedicated interpreter with access to the AppWorld APIs. In our setup, we adopt this native interpreter-based interaction protocol and use the official task definitions and evaluation harness, ensuring consistent API access and evaluation conditions across all agent configurations.

3.2. Agents

The five architectures evaluated are: ReAct, ReAct Short (ReAct with tool shortlisting), Smolagent, OpenAI Solo, and Claude Code. We treat ReAct and ReAct Short as distinct architectures because tool shortlisting is a substantive component change with measurable impact (§4.1), not a hyperparameter sweep.

ReAct We implement two ReAct-style (Yao et al., 2023) agents: a vanilla ReAct baseline using LiteLLM's (BerriAI, 2024) tool-calling interface (ReAct), and an extended version with tool shortlisting (ReAct Short). Both are integrated with Exgentic by exposing benchmark actions as tool specifications, while the shortlisting variant is designed to handle large action spaces efficiently.

Smolagent CodeAgent A code-generation agent that produces Python code to invoke tools rather than calling them directly. We integrate Smolagent v1.24.0 (Roucher et al., 2025) with Exgentic by exposing benchmark actions as Python functions and adapting its termination behavior to use the benchmark-defined finish action.

OpenAI Solo + MCP An agent built on the OpenAI Agents SDK (OpenAI, 2025a) v0.7.0 in solo mode with Model Context Protocol (Anthropic, 2024) integration (OpenAI Solo for short). The agent operates in solo mode, interacting with environments exclusively through MCP tool calls. We integrate it with Exgentic by implementing an adapter that translates benchmark actions into MCP tool specifications.

Claude Code A feature-rich command-line agent originally designed for software engineering tasks and recently claimed to be generally effective beyond coding[1]. We evaluate Claude Code v2.1.7 without modifying its internal logic, integrating it with Exgentic via MCP-exposed benchmark actions. The agent runs in a Docker container to ensure isolation and reproducibility.

3.2.1. Agent Architectural Components

Agents differ in implementation but share common conceptual components. To gain insight into agents' internal behavior and its impact on performance, we adopt a component-level view covering execution runtime, tool shortlisting, schema guards, communication protocols, memory, and planning. Appendix B.3 details their presence across agents.

3.3. Metrics

To enable consistent comparison across agents and tasks, we adopt the following general metrics.

[1] Building agents with the Claude Agent SDK.

Success Rate. The proportion of runs deemed successful according to the original success definition and evaluation procedure of the benchmark.
Cost per Task. The average monetary cost of completing a task, enabling comparison of agent efficiency in addition to performance. In our experiments, costs are reported using LiteLLM's pricing data[2].
Average Steps. The mean number of steps taken by an agent to reach task completion.
Bench-Weighted Mean. A weighted mean of success across six benchmarks, with the three τ^2-Bench subdomains aggregated and given equal total weight to each of the other benchmarks. **Cost-Efficiency.** The ratio of bench-weighted success to mean cost-per-task in dollars (score/$); component ablations report paired cost deltas at fixed (model, benchmark).

Step-Based Comparisons. For step-level analyses (§4.9), zero-step sessions (run-level orchestrator failures) are excluded (7.0% of runs) and step counts on the remaining runs are capped at 50 to limit outlier influence (10.2% of remaining runs).

Statistical Analysis. Variance decomposition uses $\eta^2 = \mathrm{Var}(\mathbb{E}[Y|X])/\mathrm{Var}(Y)$ (Cohen, 1988) with Y the cell success rate, computed across the 15 closed-source configurations. Pairwise model and architecture rankings use paired t-tests on shared (benchmark, task) outcomes; leaderboard top-vs-rest claims use a pooled McNemar test (McNemar, 1947). Step-zero rates use two-proportion z-tests; cross-benchmark agreement uses Spearman rank correlations (Spearman, 1904). Multiplicity is controlled via Benjamini–Hochberg (Benjamini & Hochberg, 1995) and Benjamini–Yekutieli (Benjamini & Yekutieli, 2001) corrections at $\alpha = 0.05$. Full procedures, p-values, confidence intervals, and bootstrap settings are in App. F.

4. Results

4.1. The Open General Agent Leaderboard

The agent-configuration leaderboard (Table 2) is led by OpenAI Solo+Claude Opus 4.5, with the top three (all Claude Opus 4.5-backed) clustering within 6pp; all top-ten configurations are closed-source pairings, with no GPT 5.2- or open-weight-backed configuration appearing. The gap from this top cluster to configurations using other backbone models is significant under cross-model paired t-tests ($p < 0.001$, App. F).

Backbone-model rankings are consistent and significant. Bench-weighted mean success rate of each model averaged across the five architectures: Claude Opus 4.5 0.66, Gemini 3 0.59, GPT 5.2 and DeepSeek-V3.2 tied at 0.41, Kimi-K2.5 0.37 ($p < 0.001$, App. F). Claude Opus 4.5-backed configurations excel broadly; the GPT 5.2-backed aggregate is dragged down by configuration failures in tool-rich environments; open-weight backbones trail the frontier on average but vary widely across architectures (§4.7).

Architecture rankings are not significant in aggregate (range ∼7pp; $p > 0.1$, App. F). But architecture-model pairings matter: OpenAI Solo+Claude Opus 4.5 reaches 0.73 while OpenAI Solo+GPT 5.2 drops to 0.39, whereas ReAct Short-based configurations perform more consistently across backbone models.

Two outliers anchor later analysis. Smolagent+Gemini 3 achieves the highest single-benchmark cell (0.88 on τ^2-Bench-Telecom). Three GPT 5.2-backed configurations score 0.00 on AppWorld without tool shortlisting because GPT 5.2's API is limited to 128 tools while AppWorld exposes ∼468 (App. A); Claude Opus 4.5-backed configurations score 0.61–0.70 on the same environment with no shortlisting, indicating the failure is a model-API limitation rather than an integration artifact (analyzed further in §4.7). Within-model architectural spread reaches 12pp for closed-source backbones and 14–18pp for open-weight backbones, with the most dramatic cell-level swing being Kimi-K2.5-backed τ^2-Bench-Telecom 0.83 vs. 0.00 across architectures. This motivates both the per-tier sensitivity analysis in §4.4 and the generality-sinks analysis in §4.7.

4.2. Generalists Match Heavily-Customized Specialists

On 4 of 6 benchmarks, the top general-agent configuration is statistically indistinguishable from the top reported domain-specific agent score (SWE-agent Team, 2024) under the $n = 100$ Wilson half-width (Wilson, 1927) of ∼8–10pp (Table 3; details in App. E.3, sources in App. E.2). On SWE-Bench Verified, τ^2-Bench-Airline, τ^2-Bench-Retail, and AppWorld, the gap is within sampling noise or favors the generalist; on BrowseComp+ and τ^2-Bench-Telecom, the generalist trails. No single architecture dominates: OpenAI Solo wins on SWE-Bench Verified and all three τ^2-Bench subdomains; Smolagent wins on AppWorld; both tie on BrowseComp+. Optimal architecture is benchmark-dependent.

4.3. Model Choice Dominates Aggregate Variance; Architecture Decides Within Cells

Within a single backbone model, architecture choice swings results by up to 12 percentage points (Claude Opus 4.5-backed configurations: best 0.73, worst 0.61). Aggregated across architectures, models, and benchmarks, model effects dwarf architecture effects by an order of magnitude (Kapoor et al., 2024; 2025). Variance explained, $\eta^2 = \mathrm{Var}(\mathbb{E}[Y|X])/\mathrm{Var}(Y)$ where Y is the cell success

[2] Model prices and context window.

Table 3. Best-performing general-agent configuration per benchmark compared against the top reported domain-specific agent performance on the original benchmark leaderboards (links: App. E.2). Scores are success rates on 100 randomly sampled instances per benchmark; original leaderboards use full benchmarks.

Benchmark	Best General-Agent Configuration	General Agent Score	Domain-Specific Score
SWE-Bench Verified	OpenAI Solo + Claude Opus 4.5	0.81	0.79
BrowseComp+	Smolagent + Claude Opus 4.5	0.61	0.80
τ^2-Bench-Airline	OpenAI Solo + Claude Opus 4.5	0.74	0.73
τ^2-Bench-Retail	OpenAI Solo + Claude Opus 4.5	0.85	0.86
τ^2-Bench-Telecom	Smolagent + Gemini 3	0.88	0.98
AppWorld	Smolagent + Claude Opus 4.5	0.70	0.73

rate and X is the grouping variable, attributes 27.8% of total variance to model choice across the 15 closed-source configurations ($p < 10^{-10}$, App. F.3); architecture explains only 0.5%. A complementary additive decomposition over (model, architecture) cell means estimates the interaction effect at 5.4%, an order of magnitude larger than architecture's main effect, indicating that optimal architecture choice depends on the model. The 58× aggregate dominance is a closed-source statistic; once the two open-weight models are included, the architecture main effect becomes detectable ($F(4, 136) = 3.82$, $p = 0.006$), reflecting the conditional architectural sensitivity examined in §4.7.

These two views are complementary, not contradictory. The aggregate η^2 averages out architecture's main effect when pooled across all dimensions; the remaining subsections show where this aggregate view conceals decisive architectural effects in specific cells.

4.4. Architectural Sensitivity Scales with Model Tier

Architecture choice matters dramatically more for open-weight than for frontier closed-source backbones (Sclar et al., 2023). Architectural spread, the difference between best and worst bench-weighted success across the five architectures within a fixed backbone model, separates the tiers: spread is 7–12pp for closed-source backbones and 14–18pp for open-weight backbones ($p < 0.001$ best-vs-worst, App. F.1). With a strong closed-source model, developers can iterate on agent design without extensive model-specific tuning; with an open-weight model, agent-architecture co-design becomes necessary, examined further in §4.7.

4.5. Cross-Benchmark Consistency

Pairwise Spearman rank correlations between benchmark scores across the 15 closed-source configurations are predominantly positive: median $+0.67$, range $[+0.44, +0.81]$ (full matrix in App. F.6). BrowseComp+ shows the lowest pairwise correlations (0.44–0.75), suggesting it captures somewhat distinct capabilities. The positive correlation structure is consistent with the variance decomposition: when a configuration ranks high on one benchmark it tends to rank high on others, leaving model identity as the dominant axis of cross-benchmark consistency (§4.3).

4.6. Schema Guards and Tool Shortlisting Yield Cross-Model Gains

Two architectural components appear to correlate with stronger performance across architectures and models. The three top-performing architectures (OpenAI Solo, Claude Code, and Smolagent) all employ a schema guard (Table 8): a mechanism that detects when an action with an invalid schema is invoked and allows the agent to self-correct. This is an observational pattern, since these architectures also differ on other components (App. B.3); the shortlisting result below is a controlled ablation. Tool shortlisting (Qin et al., 2023), when added to a simple ReAct architecture in tool-rich environments, improves performance for four of the five backbone models tested: ReAct+GPT 5.2 gains 5.5 percentage points overall, while ReAct+Claude Opus 4.5 shows a smaller gain but exhibits a $1.97 cost reduction per task. DeepSeek-V3.2 is the lone exception (App. E.4).

4.7. Generality Sinks

The two open-weight models tested (DeepSeek-V3.2, Kimi-K2.5) exhibit two distinct failure modes that frontier closed models do not (Froger et al., 2025): an architecture sink, where the same backbone model scores 0.83 with one architecture and 0.00 with another, and a benchmark sink, where every architecture collapses on a specific environment. Whether these patterns generalize beyond the two tested checkpoints is an open question.

The architecture sink concentrates on τ^2-Bench. Failures cluster on autonomous architectures (Claude Code, OpenAI Solo), whose system prompt instructs the agent to communicate exclusively through tool calls (App. B). Open-weight models violate this instruction on the first turn, emitting a direct user message instead of a tool call, and the orchestrator terminates the run immediately. The pattern is stark: 94% of Kimi-K2.5+autonomous τ^2-Bench sessions take zero steps, 31% for DeepSeek-V3.2, 1.7% for closed-

source ($p < 10^{-15}$, App. F.5). Structured architectures (ReAct, Smolagent) route output directly into tool invocations and bypass the failure, leaving open-weight models competitive: Kimi-K2.5+ReAct reaches 0.83 on τ^2-Bench-Telecom; DeepSeek-V3.2+ReAct reaches 0.82 on τ^2-Bench-Retail.

The benchmark sink appears on AppWorld: every open-weight-backed configuration collapses regardless of architecture, with the best DeepSeek-V3.2+Smolagent reaching only 0.13 versus 0.70 for Claude Opus 4.5+Smolagent, reflecting a capability ceiling for tool-rich autonomous interaction over ∼468 actions. GPT 5.2-backed configurations show a related collapse with a different mechanism: GPT 5.2's 128-tool API limit on AppWorld's ∼468 actions drives three of four GPT 5.2 configurations to 0.00, with tool shortlisting recovering only to 0.22 (§4.1).

τ^2-Bench exhibits the architecture sink, AppWorld the benchmark sink; SWE-Bench Verified and BrowseComp+ exhibit neither (gap 95% bootstrap CIs in App. F.4).

4.8. Cost-Efficiency Spans 30× Without a Universal Best

Cost-efficiency (success per dollar of inference cost; per-task cost methodology in App. G.4) varies by approximately 30× across agent configurations (Fig. 1; full breakdown in App. E.1). GPT 5.2- and DeepSeek-V3.2-backed configurations share the efficiency frontier (ReAct+GPT 5.2 and OpenAI Solo+DeepSeek-V3.2 tie at 2.43 score/$), while the highest-scoring configuration (OpenAI Solo+Claude Opus 4.5, 0.73 score) operates at 0.09 score/$ and the least efficient (Claude Code+Claude Opus 4.5) at 0.08 score/$.

4.9. Failure Patterns Expose Architectural Differences in Resource Use

Failed runs are systematically more expensive than successful ones, and architectures differ in how they fail. Comparing successful and failed runs at the task level (across the three closed-source backbones — open-weight autonomous failures terminate by protocol violation rather than resource exhaustion, §4.7 — after excluding zero-step sessions and capping step counts at 50), bench-weighted averages are positive for every architecture (Claude Code +39%, OpenAI Solo +20%, Smolagent +26%, ReAct +54%, ReAct Short +45%). The largest overheads appear on interaction-heavy benchmarks (ReAct on AppWorld shows +111%, Claude Code on BrowseComp+ shows +70%); a few τ^2-Bench cells are near zero or negative, indicating early-termination failure modes. The step-count data describes how much each architecture spends on failed runs but does not isolate the underlying mechanism. Detailed per-cell breakdowns are in App. E.5 and App. E.6.

4.10. Behavioral Failure Analysis

Step counts describe *how* architectures fail; failure modes describe *why*. We adapt ErrorMap (Ashury-Tahan et al., 2026) to the 2,868 failed sessions with full trajectories, producing 27 failure categories (App. H). Although architecture contributes only 0.5% to success-rate variance (§4.3), each architecture has a distinctive failure profile: Claude Code and OpenAI Solo over-represent *Premature Termination*; ReAct and ReAct Short over-represent *Evidence Retrieval Omission*; and Smolagent is dominated by *Search Recovery & Adaptation* (Fig. 2). A chi-squared test of the architecture × failure-category contingency table rejects the architecture-uniform null at $\chi^2(196) = 328.5$, $p < 10^{-10}$ (Cramér's $V = 0.17$); failure modes thus discriminate architectures that aggregate scoring cannot. Per-model, per-architecture, and per-benchmark breakdowns and judge validation are in App. H.

5. Related Work

Domain-Specific Agent Benchmarks. The rapid advancement of AI agents has led to a proliferation of benchmarks (Zhou et al., 2023; Deng et al., 2023; Xie et al., 2024; Liu et al., 2023; Mialon et al., 2023), each targeting specific domains such as software engineering (Jimenez et al., 2023; Yang et al., 2024; Merrill et al., 2026), customer service (Yao et al., 2024), and deep scientific research (Bragg et al., 2025). Each benchmark defines domain-specific protocols and task specifications.

Attempts at Consolidation. HAL (Kapoor et al., 2025) unifies infrastructure across benchmarks but requires per-benchmark agent adaptation. BrowserGym (Chezelles et al., 2025) and Harbor (Shaw, 2025) standardize interaction via fixed protocols (web/CLI) but restrict evaluation to single environment classes. Inspect (AI Security Institute, UK, 2024) consolidates the *infrastructure* layer (sandboxing, logging, scoring, and native agent execution, with log-analysis tools such as Scout built on top of it), but its `Task(solver=[use_tools([...]), agent()])` pattern still requires the evaluation author to manually choose a tool-set, sandbox, and solver per benchmark. AgentBeats[3] requires agents and benchmarks to conform to A2A+MCP subsets for evaluation; CUBE (Lacoste et al., 2026) unifies benchmarks via an MCP+Gym schema (wrap once, consume universally). Both bind one or both sides to a specific transport; the Unified Protocol is instead a thin mediation layer that wraps existing agents and benchmarks in their native protocols, without rewriting either side. The Unified Protocol operates at the *integration* layer, mediating between agent protocols and benchmark interfaces via a canonical task/context/actions representa-

[3] AgentBeats

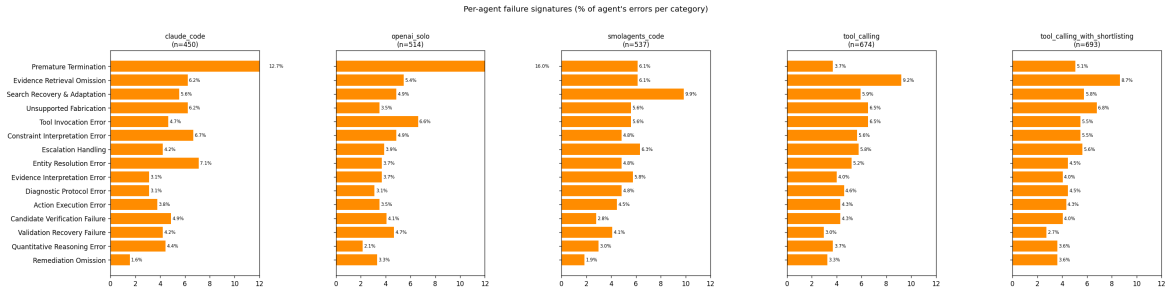

Figure 2. Per-architecture failure-category shares (top-15 categories, column-normalized). The architecture axis produces structured spread on this dependent variable, in contrast to its 0.5% contribution to success-rate variance (§4.3); per-architecture findings are discussed in §4.10.

tion. Exgentic implements the Unified Protocol and can be layered on top of infrastructure harnesses like Inspect, enabling protocol-preserving evaluation across heterogeneous benchmarks without per-benchmark agent adaptation. A side-by-side comparison of evaluation systems across five axes is in Table 1.

6. Discussion

This work takes a step toward the systematic study of general-purpose agents, a target the community has called for but not yet pursued. We develop Exgentic and the Unified Protocol as infrastructure that lets unmodified agents run on unmodified benchmarks, and release the first Open General Agent Leaderboard as a public reference for measuring progress.

Across the configurations we evaluate, general agents reach meaningful performance on every domain without any per-domain customization (§4.1). On most benchmarks, the best general-agent configuration is indistinguishable from heavily-customized domain specialists (§4.2), showing that general agents are a viable research direction.

The leaderboard analysis also points to clear directions for progress. Models need to be trained to work consistently across architectures (§4.3), so that model research and agent-architecture research can advance independently. Today, the same open-weight model can swing between strong performance and total failure depending on which architecture it pairs with, entangling the two research lines (§4.7). New agent architectures and components are worth pursuing: simple components such as schema guards and tool shortlisting moved performance meaningfully across models, and the behavioral failure analysis (§4.10) points to additional architecture-specific targets (e.g., resisting early termination), together suggesting substantial headroom (§4.6). Cost per task also varies widely across configurations, leaving cost-efficiency as a third open axis (§4.8). The Open General Agent Leaderboard provides infrastructure for tracking progress on each of these axes; we invite the community to extend it with new agents, benchmarks, and protocol adaptors as they emerge.

Limitations. Our evaluation is bounded by cost and scope: five LLMs, five agent implementations, and six benchmarks at ∼$20K. Per-benchmark scores carry Wilson CI half-widths of ±7–9.5pp; aggregated comparisons remain highly significant (App. F). Multimodal and continuous-action extensions are discussed in App. I.

References

AI Security Institute, UK. Inspect AI: Framework for large language model evaluations, 5 2024. URL https://github.com/UKGovernmentBEIS/inspect_ai.

Anthropic. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, November 2024. Specification: https://modelcontextprotocol.io.

Anthropic. System card: Claude Opus 4.5. https://www.anthropic.com/news/claude-opus-4-5, November 2025. Model identifier: claude-opus-4-5.

Ashury-Tahan, S., Mai, Y., Bandel, E., Shmueli-Scheuer, M., and Choshen, L. ErrorMap and ErrorAtlas: Charting the failure landscape of large language models, 2026. URL https://arxiv.org/abs/2601.15812.

Bandel, E., Yehudai, A., Lacoste, A., Ghosh, A., Neubig, G., Mitchell, M., Shmueli-Scheuer, M., and Choshen, L. Position: Agentic systems should be general. *SSRN Electronic Journal*, February 2026a. doi: 10.2139/ssrn.6176178. URL https://ssrn.com/abstract=6176178.

Bandel, E., Yehudai, A., and Shmueli-Scheuer, M. Ready for general agents? let's test it. In *ICLR Blogposts 2026*, 2026b. URL https:

//iclr-blogposts.github.io/2026/blog/2026/general-agent-evaluation/.

Benjamini, Y. and Hochberg, Y. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society: Series B (Methodological)*, 57(1):289–300, 1995. doi: 10.1111/j.2517-6161.1995.tb02031.x.

Benjamini, Y. and Yekutieli, D. The control of the false discovery rate in multiple testing under dependency. *The Annals of Statistics*, 29(4):1165–1188, 2001. doi: 10.1214/aos/1013699998.

BerriAI. LiteLLM: Call all LLM APIs using the OpenAI format. https://github.com/BerriAI/litellm, 2024.

Bragg, J., D'Arcy, M., Balepur, N., Bareket, D., Dalvi, B., Feldman, S., Haddad, D., Hwang, J. D., Jansen, P., Kishore, V., et al. Astabench: Rigorous benchmarking of ai agents with a scientific research suite. *arXiv preprint arXiv:2510.21652*, 2025.

Chen, Z., Ma, X., Zhuang, S., Nie, P., Zou, K., Liu, A., Green, J., Patel, K., Meng, R., Su, M., Sharifymoghaddam, S., Li, Y., Hong, H., Shi, X., Liu, X., Thakur, N., Zhang, C., Gao, L., Chen, W., and Lin, J. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL https://arxiv.org/abs/2508.06600.

Chezelles, T. L. S. D., Gasse, M., Drouin, A., Caccia, M., Boisvert, L., Thakkar, M., Marty, T., Assouel, R., Shayegan, S. O., Jang, L. K., Lù, X. H., Yoran, O., Kong, D., Xu, F. F., Reddy, S., Cappart, Q., Neubig, G., Salakhutdinov, R., Chapados, N., and Lacoste, A. The browsergym ecosystem for web agent research, 2025. URL https://arxiv.org/abs/2412.05467.

Cohen, J. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates, Hillsdale, NJ, 2nd edition, 1988.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025. URL https://arxiv.org/abs/2512.02556.

Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., and Su, Y. Mind2web: Towards a generalist agent for the web. *Advances in Neural Information Processing Systems*, 36:28091–28114, 2023.

Froger, R., Andrews, P., Bettini, M., Budhiraja, A., Cabral, R. S., Do, V., Garreau, E., Gaya, J.-B., Laurençon, H., Lecanu, M., Malkan, K., Mekala, D., Ménard, P., Bertran, G. M.-T., Piterbarg, U., Plekhanov, M., Rita, M., Rusakov, A., Vorotilov, V., Wang, M., Yu, I., Benhalloum, A., Mialon, G., and Scialom, T. Are: Scaling up agent environments and evaluations, 2025. URL https://arxiv.org/abs/2509.17158.

Google DeepMind. Gemini 3: Introducing the latest Gemini AI model from Google. https://blog.google/products-and-platforms/products/gemini/gemini-3/, November 2025. Model identifier: gemini-3-pro-preview.

Jimenez, C. E., Yang, J., Wettig, A., Yao, S., Pei, K., Press, O., and Narasimhan, K. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Kapoor, S., Stroebl, B., Siegel, Z. S., Nadgir, N., and Narayanan, A. AI agents that matter. *CoRR*, abs/2407.01502, 2024. doi: 10.48550/ARXIV.2407.01502. URL https://doi.org/10.48550/arXiv.2407.01502.

Kapoor, S., Stroebl, B., Kirgis, P., Nadgir, N., Siegel, Z. S., Wei, B., Xue, T., Chen, Z., Chen, F., Utpala, S., Ndzomga, F., Oruganty, D., Luskin, S., Liu, K., Yu, B., Arora, A., Hahm, D., Trivedi, H., Sun, H., Lee, J., Jin, T., Mai, Y., Zhou, Y., Zhu, Y., Bommasani, R., Kang, D., Song, D., Henderson, P., Su, Y., Liang, P., and Narayanan, A. Holistic agent leaderboard: The missing infrastructure for ai agent evaluation, 2025. URL https://arxiv.org/abs/2510.11977.

Kimi Team. Kimi K2.5: Visual agentic intelligence, 2026. URL https://arxiv.org/abs/2602.02276.

Lacoste, A., Gontier, N., Shliazhko, O., Jaiswal, A., Sareen, K., Nanisetty, S., Cabezas, J., Verme, M. D., Younis, O. G., Baratta, S., Avalle, M., Kerboua, I., Lù, X. H., Bandel, E., Shmueli-Scheuer, M., Yehudai, A., Choshen, L., Lebensold, J., Hughes, S., Caccia, M., Drouin, A., Reddy, S., Yu, T., Su, Y., Neubig, G., and Song, D. Cube: A standard for unifying agent benchmarks, 2026. URL https://arxiv.org/abs/2603.15798.

Liu, X., Yu, H., Zhang, H., Xu, Y., Lei, X., Lai, H., Gu, Y., Ding, H., Men, K., Yang, K., Zhang, S., Deng, X., Zeng, A., Du, Z., Zhang, C., Shen, S., Zhang, T., Su, Y., Sun, H., Huang, M., Dong, Y., and Tang, J. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Marreed, S., Oved, A., Yaeli, A., Shlomov, S., Levy, I., Akrabi, O., Sela, A., Adi, A., and Mashkif, N. Towards enterprise-ready computer using generalist agent, 2025. URL https://arxiv.org/abs/2503.01861.

McNemar, Q. Note on the sampling error of the difference between correlated proportions or percentages. *Psychometrika*, 12(2):153–157, 1947. doi: 10.1007/BF02295996.

Merrill, M. A., Shaw, A. G., Carlini, N., Li, B., Raj, H., Bercovich, I., Shi, L., Shin, J. Y., Walshe, T., Buchanan, E. K., et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

Mialon, G., Fourrier, C., Swift, C., Wolf, T., LeCun, Y., and Scialom, T. GAIA: a benchmark for General AI Assistants. *arXiv preprint arXiv:2311.12983*, 2023.

OpenAI. OpenAI Agents SDK. https://openai.github.io/openai-agents-python/, 2025a. Repository: https://github.com/openai/openai-agents-python.

OpenAI. Update to GPT-5 system card: GPT-5.2. https://openai.com/index/gpt-5-system-card-update-gpt-5-2/, December 2025b. System card; gpt-5.2-2025-12-11.

Perlitz, Y., Bandel, E., Gera, A., Arviv, O., Ein-Dor, L., Shnarch, E., Slonim, N., Shmueli-Scheuer, M., and Choshen, L. Efficient benchmarking of language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2024.

Qin, Y., Liang, S., Ye, Y., Zhu, K., Yan, L., Lu, Y., Lin, Y., Cong, X., Tang, X., Qian, B., Zhao, S., Tian, R., Xie, R., Zhou, J., Gerstein, M., Li, D., Liu, Z., and Sun, M. Toolllm: Facilitating large language models to master 16000+ real-world apis, 2023.

Robertson, S. E., Walker, S., Jones, S., Hancock-Beaulieu, M. M., and Gatford, M. Okapi at trec-3. In *Proceedings of the Third Text REtrieval Conference (TREC-3)*, pp. 109–126. NIST, 1994.

Roucher, A., del Moral, A. V., Wolf, T., von Werra, L., and Kaunismäki, E. 'smolagents': a smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025.

Sclar, M., Choi, Y., Tsvetkov, Y., and Suhr, A. Quantifying language models' sensitivity to spurious features in prompt design or: How i learned to start worrying about prompt formatting. *arXiv preprint arXiv:2310.11324*, 2023.

Shaw, A. Harbor Framework, November 2025. URL https://github.com/laude-institute/harbor.

Spearman, C. The proof and measurement of association between two things. *The American Journal of Psychology*, 15(1):72–101, 1904. doi: 10.2307/1412159.

SWE-agent Team. mini-SWE-agent: A minimalist agent for SWE-bench and beyond. https://github.com/SWE-agent/mini-swe-agent, 2024.

Wilson, E. B. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927. doi: 10.1080/01621459.1927.10502953.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., and Yu, T. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, 2024.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. SWE-agent: Agent-computer interfaces enable automated software engineering. In *Advances in Neural Information Processing Systems*, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023.

Yao, S., Shinn, N., Razavi, P., and Narasimhan, K. Taubench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Yehudai, A., Eden, L., Li, A., Uziel, G., Zhao, Y., Bar-Haim, R., Cohan, A., and Shmueli-Scheuer, M. Survey on evaluation of llm-based agents, 2025. URL https://arxiv.org/abs/2503.16416.

Zhang, Y., Ruan, H., Fan, Z., and Roychoudhury, A. Autocoderover: Autonomous program improvement. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 1592–1604, 2024.

Zhang, Y., Li, M., Long, D., Zhang, X., Lin, H., Yang, B., Xie, P., Yang, A., Liu, D., Lin, J., Huang, F., and Zhou, J. Qwen3 embedding: Advancing text embedding and reranking through foundation models, 2025. URL https://arxiv.org/abs/2506.05176.

Zhou, S., Xu, F. F., Zhu, H., Zhou, X., Lo, R., Sridhar, A., Cheng, X., Ou, T., Bisk, Y., Fried, D., Alon, U., and Neubig, G. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

Appendix Contents

A Framework and Adaptation Details 13

B Agent Adaptation 17

C Benchmark Adaptation 19

D Detailed Benchmark Agent Interaction Example 25

E Detailed Results 26

F Statistical Significance 29

G Reproducibility Details 35

H Behavioral Error Analysis on Agentic Trajectories 42

I Limitations 46

A. Framework and Adaptation Details

This appendix details how we adapt existing benchmarks and agents to the Unified Protocol, and the Exgentic orchestrator design that runs the full factorial evaluation.

Table 4 reproduces the comparison table from the main paper (Table 1) with the full caption defining each of the five axes used to position this work against prior evaluation systems.

Table 4. Positioning of this work relative to prior agent-evaluation studies and frameworks along five axes. ✓ indicates the property holds; ✗ indicates it does not; ✓× indicates it holds within a constrained class. *Multi-Protocol Benchmark*: benchmarks are supported in their native form across heterogeneous protocols, without forcing a fixed transport. *Multi-Protocol Agent*: agents are supported in their native form across heterogeneous protocols. *Agent Plug-and-Play*: an agent integrates without per-benchmark adaptation. *General-Purpose Tasks*: evaluated tasks span domains rather than a single application class. *Systematic Factorial Study*: the work performs a comparative study that decomposes performance across the full agent-architecture × backbone-model × benchmark factorial. This is the extended version of Table 1 in the main paper.

System	Multi-Protocol Benchmark	Multi-Protocol Agent	Agent Plug-and-Play	General-Purpose Tasks	Systematic Factorial Study
agent-eval (Bragg et al., 2025)	✓ (Inspect harness)	✓ (InspectAI solvers)	✗ (solver implementation)	✗ (scientific research only)	✓× (no model axis; sci only)
HAL (Kapoor et al., 2025)	✓	✓	✗ (per-benchmark adaptation)	✓	✓× (architecture per-benchmark wired)
Inspect (AI Security Institute, UK, 2024)	✓	✓	✗ (per-task solver setup)	✓	✗ (framework only)
BrowserGym (Chezelles et al., 2025)	✗ (web only)	✗ (custom API)	✓	✓× (web-mediated)	✓× (single architecture across LLMs)
Harbor (Shaw, 2025)	✗ (CLI only)	✗ (CLI only)	✓	✓× (CLI-mediated)	✗ (framework only)
AgentBeats	✗ (A2A+MCP)	✗ (A2A+MCP)	✓	✓	✗ (framework only)
CUBE (Lacoste et al., 2026)	✗ (MCP+Gym)	✗ (MCP+Gym)	✓	✗	✗ (position paper; no empirical eval)
This work	✓	✓	✓	✓	✓

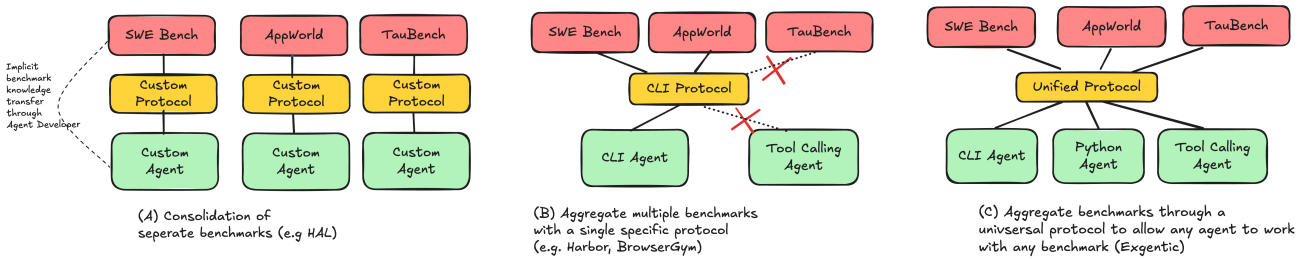

Figure 3. Evolution of Agentic Evaluation. (A) Collection of separate benchmarks, each requiring a custom agent or an agent with specific adaptation per benchmark (HAL). (B) Multiple benchmarks consolidated through a single protocol, such as CLI or Web. (C) Multiple benchmarks consolidated through a common protocol that can be adapted to any agent's protocol (Exgentic).

A.1. Adapting Existing Benchmarks

Existing agent benchmarks are typically coupled with specific interaction protocols, and often implicitly assume that agents possess prior knowledge of the benchmark's semantics, or that a human will manually perform the integration.

A representative example is SWE-BENCH VERIFIED[4]. Each task specifies a GitHub repo, a base commit, and a free-text bug description, with the expected output being a patch. The benchmark does not define how agents should access the repo or submit fixes; those details are left to the integrator. For general-purpose agents without human intervention, this interface must be explicit. However, we cannot arbitrarily decide on a setup; instead, we derive the interface from a reference agent implementation.

For SWE-BENCH VERIFIED, we examined MINI-SWE AGENT[5] as the reference implementation. There, the agent is placed in a bash environment where the repository has already been cloned. When the agent outputs `COMPLETE_TASK_AND_SUBMIT_FINAL_OUTPUT`, the system automatically generates a patch and submits it for evaluation. This design fully specifies how the agent interacts with the benchmark, what actions it may take, and how it submits solutions, implicitly indicating that repository cloning and patch creation are *not* evaluation targets.

[4] SWE-Bench Verified
[5] MINI-SWE AGENT

Accordingly, in the Exgentic protocol for SWE-BENCH VERIFIED, we introduce two explicit actions: one for executing bash commands and another for submitting a patch constructed from the agent's code modifications.

To define the protocol's task and context fields, we review both the benchmark tasks and the reference implementation prompts. Many benchmark tasks include irrelevant implementation details, while key instructions appear only in the reference agent's internal prompts. For instance, in τ^2-Bench, the reference prompt states: *"You are a customer service agent that helps the user according to the `<policy>` below."* Such essential information belongs in the benchmark task itself and is included in the Exgentic task definition. In contrast, instructions like *"Each turn you may either message the user or make a tool call, but not both"* are excluded because they assume a particular tool-calling protocol.

In summary, we decouple each benchmark from its original protocol by making all agent-visible assumptions explicit. First, we inspect the reference agent to see how it interacts with the environment and what actions and observations it uses. Then we build task descriptions that include only the information needed for the agent to solve the task, omitting implementation-specific details and redundant signals. This yields tasks that preserve the benchmark's intended semantics while remaining independent of any particular agent architecture or communication protocol, making them suitable for evaluating any general agent implementation.

A.1.1. PER-BENCHMARK ADAPTATIONS

Table 5 summarizes the six adaptations along the three protocol fields plus the reference implementation we mirrored. The full task / context / actions text for each benchmark is in Appendix C; the released adaptor code lives at `src/exgentic/benchmarks/<benchmark>/` in the released repository.

Benchmark	Reference impl.	Actions exposed	Task / context fields
SWE-Bench Verified	mini-SWE agent	`bash`, `finish`	Task: bash-protocol prompt + issue body. Context: empty.
AppWorld	native interpreter	one per AppWorld API (∼468); `finish`	Task: supervisor instruction. Context: `policy`, `supervisor`, `datetime`.
BrowseComp+	authors' retriever	`search`, `get_document`, `submit`	Task: scoring-rubric prompt + question. Context: empty.
τ^2-Airline	τ^2 reference ReAct	`message` + benchmark tools	Task: "customer service agent" line. Context: `policy` (airline).
τ^2-Retail	τ^2 reference ReAct	`message` + benchmark tools	As above; `policy` (retail).
τ^2-Telecom	τ^2 reference ReAct	`message` + benchmark tools	As above; `policy` (telecom).

Table 5. Per-benchmark adaptation summary. Released adaptor code: `src/exgentic/benchmarks/{appworld, browsecompplus, swebench, tau2}/`.

Table 6 pins the canonical source for each reference implementation we mirrored.

Benchmark	Reference agent	Source
SWE-Bench Verified	mini-SWE agent	github.com/SWE-agent/mini-swe-agent
AppWorld	`minimal_agent.ipynb` (ReAct)	StonyBrookNLP/appworld/notebooks/minimal_agent.ipynb
BrowseComp+	OpenAI search-agent baseline	texttron/BrowseComp-Plus/search_agent/openai_client.py
τ^2-Bench	`LLMAgent` (text default)	sierra-research/tau2-bench/src/tau2/agent/llm_agent.py

Table 6. Reference agent implementations mirrored when defining each benchmark's Unified Protocol adaptor. Each row links to the canonical implementation in the benchmark's own repository.

AppWorld. The native interface is a dedicated Python interpreter exposing the AppWorld APIs as importable functions; the reference agent writes Python code that the interpreter executes. We preserve this interface by treating each AppWorld API (∼468 across the nine apps) as a distinct Unified Protocol action, generated programmatically from `world.task.api_docs.function_calling()`. The benchmark's native completion call (`supervisor.complete_task`) becomes the protocol's `finish` action. The task field carries the supervisor's natural-language instruction (`"Task from supervisor:\n{instruction}"`); environment-level information

that the reference agent assumes implicitly — which apps are available, how supervisor credentials are obtained, time and pagination conventions — is moved into the context field as a `policy` entry, alongside the `supervisor` record and the simulated `datetime`. Code-generation agents are not privileged: any agent that can call the exposed actions (Python functions, tool-calls, MCP) can solve AppWorld tasks. Because GPT 5.2's tool-use API enforces a 128-tool ceiling per request, GPT 5.2-backed configurations without tool shortlisting cannot expose the full 468 actions and score 0.00 on AppWorld; this is a model-API limitation rather than an integration artifact (§4.1, §4.7).

BrowseComp+. BrowseComp+ provides queries, a fixed corpus, and an authors' retriever; the reference harness expects an agent that can issue searches and (optionally) fetch full documents. We expose three actions: `search` (single `query` string, returns the top-k snippets each trimmed to a documented token budget), an optional `get_document` (full text by document id), and `submit` (finish action, with structured fields `exact_answer`, `explanation`, `confidence`). The task field embeds the question along with the benchmark's scoring rubric and a hard rule that finishing requires `submit` (since the original benchmark's user interaction does not exist in our setup). Context is empty: the corpus is server-side, accessed only through `search`/`get_document`.

τ^2-Bench (Airline / Retail / Telecom). The native interface is the bespoke Python API in τ^2-Bench: the agent receives messages from a simulated user and either replies with a message or invokes one of the benchmark-registered tools (e.g., `cancel_reservation`, `search_direct_flight`). We preserve this by mapping the user-reply path to the protocol's `message` action, and translating each τ^2 tool (via its OpenAI schema) into a distinct Unified Protocol action. The task field is the single line *"You are a customer service agent that helps the user according to the `<policy>` provided below. Try to be helpful and always follow the policy."* The domain-specific policy (airline / retail / telecom rules) is exposed through the context field. Tool-calling-protocol-specific instructions present in the reference agent (e.g., "each turn you may either message the user or make a tool call, but not both") are deliberately omitted, because they constrain protocol shape rather than task semantics.

A.2. Adapting Existing Agents

Existing agents interface with existing environments through specific protocols such as MCP, Python functions, or tool calls. They also receive the task description through some command line or programmatic API.

Adapting agents to the Unified Protocol involves deciding how to map the task, context, and actions of the protocol to the agents' specific API. It is important to note that the agent adaptor is benchmark agnostic.

The textual task descriptions are typically concatenated with the context fields into textual instructions passed to the model. While not implemented today, the context may be used in different ways. For example, an MCP-based agent may opt to store the context in MCP resources rather than add them to the instructions.

Action adaptation is straightforward and largely reusable across agents using similar APIs, with each action mapped to a single Python function, OpenAI tool, or MCP tool.

More subtle adaptation is dealing with special actions. One special action type is interacting with a user. Some agents, like tool-calling agents, natively interact with users using dedicated *assistant*- and *user*- messages rather than through tool API. To preserve the principle of presenting the benchmark to the agent in the most natural way, the tool-calling agent adaptor converts *user* and *assistant* messages to the corresponding *message* action.

A.3. Exgentic Orchestrator Design

General-purpose agents must operate across diverse environments, and hence viable evaluation frameworks must scale across many benchmarks and agents. The Exgentic framework enables running any currently supported agent on any supported benchmark task, with any LLM, using only a few lines of standard Python code or a dedicated GUI.

The framework was built for use at scale and supports parallelism and caching. Every run is executed in an isolated environment and is reproducible. Benchmark results, interaction trajectories, and cost reports are created in a standardized format for all benchmarks and agents.

The main orchestration loop is illustrated in Figure 4. Each benchmark generates a set of sessions, where each session corresponds to a single benchmark task the agent must complete (e.g., resolving a GitHub issue, or fulfilling a specific user request). For each session, the orchestrator initializes the agent with the task description, contextual information, and the

set of available actions.

Following initialization, the agent receives the first observation from the session environment and responds by selecting one of the permissible actions. This action is executed by the environment, which returns a new observation. The loop continues until either the session concludes or the agent terminates by emitting no further actions. We also terminate if the number of actions/observations exceeds some threshold to avoid deadlocks or excessive costs.

A.3.1. Solving the Integration Problem

Adapting existing agents and benchmarks to the Unified Protocol and integrating them with the Exgentic orchestrator is conceptually straightforward but practically challenging. These components are developed independently by third-party authors who are unaware of the Unified Protocol, the orchestrator's execution model, or each other's design assumptions.

Presumably, one possible solution is to make intrusive modifications to the benchmark and agent code bases to make them use the Unified Protocol. However, such changes may be extremely costly to implement, difficult to maintain, or even impossible when the agent or benchmark is closed-source.

Instead, we use external adaptor code that handles synchronization and protocol translation. On the agent side, adaptors expose the Unified Protocol actions in whatever form the agent expects (Python functions, MCP server tools, or OpenAI tools). On the benchmark side, they translate each benchmark's task definition and agent interface into the Unified Protocol. Since many adaptations repeat across agents and benchmarks, we provide base adaptors that simplify building specific ones.

We allow agents and benchmarks to run natively and independently in separate processes, while all communication between them is mediated by the orchestrator and the corresponding adaptor components. This design ensures that neither the benchmarks nor the agents are affected by the fact that they are running inside the Exgentic framework, preserving their original behavior.

For a complete walkthrough of an interaction between a code-generation agent (Smolagent) and τ^2-Bench, see Appendix D.

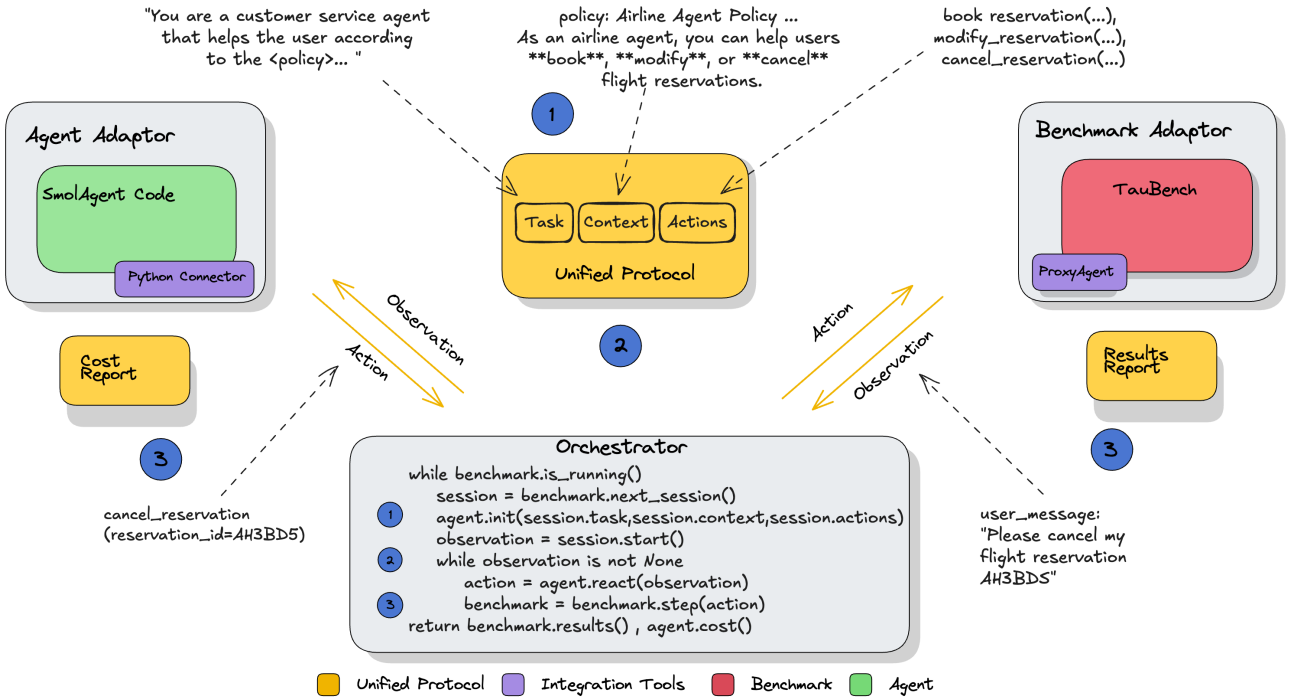

Figure 4. Exgentic architecture. Exgentic defines a unified protocol between agents and benchmarks. The Exgentic Orchestrator connects the agent and the benchmark, first passing the task definition and then mediating the observations and actions that are passed between the benchmark and the agent. Exgentic provides adaptors that convert the Unified Protocol into the specific protocols required by the agents and benchmarks. Finally, the benchmark provides the quality result metrics while the agent provides the agent runtime cost.

B. Agent Adaptation

This appendix documents the five agent integrations behind the Open General Agent Leaderboard: pinned versions, the system prompt / first-turn instructions each agent receives, and the integration approach. The released agent adaptor code lives at `src/exgentic/agents/<agent>/` in the released repository.

B.1. Pinned Agent Versions

Table 7 pins the exact version string used for each agent in every cell of the leaderboard; the same strings appear in the agent column of the released `results.csv`.

Agent	Version	Source
ReAct	`exgentic_0.1.0`	first-party (LiteLLM tool-calling)
ReAct Short	`exgentic_0.1.0` + `litellm_1.79.1`	first-party + tool shortlisting
Smolagent	`smolagents_1.24.0`	HuggingFace `smolagents`
OpenAI Solo	`openai_sdk_0.7.0`	`openai-agents` (Solo + MCP)
Claude Code	`claude_code_2.1.7`	`@anthropic-ai/claude-code` (Docker)

Table 7. Agent versions used in every leaderboard cell. Versions for ReAct/ReAct Short reflect the first-party adaptor's release; the ReAct Short variant additionally pins the LiteLLM tool-calling library it uses for shortlisting. Claude Code runs unmodified inside a Docker container that pins the CLI version via `@anthropic-ai/claude-code@2.1.7`.

B.2. First-Turn Instructions per Agent

The Unified Protocol provides each agent with three protocol fields (task, context, actions). Each agent adaptor decides how to present these fields to the underlying agent. Most adaptors prepend a small agent-specific instruction string to make the protocol's expectations explicit (e.g., that the agent is in solo mode and must finish via a designated action); the strings are short and stable across all benchmarks. We reproduce them verbatim below.

ReAct / ReAct Short (LiteLLM tool-calling). No agent-side instruction is added. The first user message simply concatenates the task and the context dictionary, with each context field tagged by an XML-style key:

```
{task}
<context_key_1>
{context_value_1}
</context_key_1>
<context_key_2>
...
```

Actions are exposed as standard tool-call specifications via LiteLLM.

ReAct Short: tool-shortlisting prompt. The ReAct Short variant adds a per-turn shortlisting step before each main call: when the benchmark exposes more than $k = 30$ actions, the same backbone LLM is queried with the conversation-so-far and the full action catalog and asked to return the k most relevant tools as a JSON list; only those k tools are passed to the next main call. If at most k actions are available, the step is bypassed and all tools are forwarded unchanged. The shortlisting call uses the following two-message prompt (`developer` role + `user` role); `{tool_catalog}` is the bullet-listed `name: description` for every available action, and `{history}` is a plain-text rendering of the conversation so far:

```
[developer]
Please before providing your next move list the names of the top 30 tools that are
somewhat relevant for the next step, ordered by relevancy (most to least). Return ONLY a
 JSON object with this shape: {
  "tools": ["tool_name_1", "tool_name_2", ...]
}.
Choose from these tools only: {tool_catalog}.
Do not call any of those tools just return the list of the top 30 relevant tools names
in the required format.
```

```
[user]
Conversation so far (plain text):
{history}
```

A high-level component view of shortlisting is in §B.3.

Smolagent (Smolagent code agent). Actions are exposed as Python functions. The first user message is:

```
Task: {task}

Context: {context}

Complete this task using the available functions. Each function corresponds to
an action you can take to solve the given task. Every action should be taken only
by calling one of the functions. If one function fail, consider using another,
at any given point one of the functions can be a valid next step. At any point
you should executing actions by writing code. do not call tools with tool calling
mechanism.

Printing or any other code will be visible only by you alone.
```

The built-in `final_answer` tool is removed so the agent terminates only via the benchmark-defined finish action.

OpenAI Solo (MCP solo mode). Actions are exposed as MCP tools. The first user message is:

```
Context: {context}

Complete this task using the available tools. Each tool corresponds to an action
you can take in the environment. Do not respond or ask clarification questions
unless done through a dedicated tool, and only if such tool exist. Any plain
message that is not a tool call will end the run in failure.
{task}
```

The agent runs in OpenAI's documented "solo mode" so all agent–environment interactions go through MCP tool calls.

Claude Code (Claude Code CLI). Actions are exposed as MCP tools through a sidecar server; the CLI runs unmodified inside a Docker container. The first user message is:

```
Context: {context}

Complete this task using the available environment tools. Each tool corresponds
to an action you can take in the task environment.
# Important: You are on solo mode. Do not reply back or message unless its
through a dedicated environment tool call, every such attempt will finish the
session with failure.
All your actions on with regard to the task must go through environment tool
calls.
Use the designated finish tool(s): {finish_tool_names}. Always conclude by
invoking the designated finish tool for this task environment.
{task}
```

Action translation. Each protocol action is mapped to a Python function (Smolagent), an OpenAI tool spec (ReAct, ReAct Short), or an MCP tool (OpenAI Solo, Claude Code); special `message` and `finish` actions follow the methodology in Appendix A.2.

B.3. Architectural Components

We outline the key components common to general-purpose agents and, in Table 8, summarize which components each evaluated agent provides.

Execution Runtime. Agents may have access to sandboxed execution environments where they can run code dynamically. For example, SmolAgents provides a Python interpreter, while Claude Code operates within a Linux machine environment. These runtime environments enable agents to execute and test code as part of their problem-solving process.

Tool Shortlisting. A preprocessing component that filters the available tool set before each action step, selecting a relevant subset based on current context. This improves efficiency and decision quality by focusing the agent on contextually appropriate tools, and addresses LLM constraints on tool count: when the full tool set exceeds model limits, shortlisting becomes necessary for task completion.

Tool Schema Guard. A component that validates actions against expected schemas before execution. When an agent attempts to call a tool or execute an environment action with incorrect parameters or structure, the schema validator raises an internal error, allowing the agent to detect and correct the mistake. This component is implemented differently across agent types: tool-calling agents typically lack explicit schema validation (relying on the LLM to generate correct calls), MCP-based agents include built-in schema validation as part of the protocol, and Python-based agents receive runtime errors from the interpreter that serve a similar validation function.

Communication Protocol. The interface through which agents invoke tools and receive results. Agents may use direct tool-calling APIs (e.g., OpenAI function calling), code-generation approaches where the agent writes executable code, or standardized protocols like MCP. Protocol choice affects action expressiveness and error handling mechanisms available to the agent.

Memory. Explicit storage and retrieval mechanisms beyond the conversation history. Memory components allow agents to maintain working state across turns, recall previous observations, and avoid redundant actions. Without explicit memory, agents rely solely on the LLM's context window.

Planning. Components that decompose tasks into structured subgoals before execution. Planning modules may generate explicit task hierarchies or action sequences, enabling more directed problem-solving. Agents without planning components select actions reactively at each step based on immediate observations.

Table 8. Architectural components of evaluated agents. ✓ denotes an explicit, modular component; ✓× denotes an implicit or non-modular capability; ✗ denotes absence.

Agent	Execution Runtime	Tool Shortlisting	Tool Schema Guard	Communication Protocol	Memory	Planning
ReAct	✗	✗	✗	Tool-calling	✓×	✓×
ReAct Short	✗	✓	✗	Tool-calling	✓×	✓×
Smolagent	✓	✗	✓	Python-Functions	✓×	✓×
OpenAI Solo	✗	✗	✓	MCP	✓×	✓×
Claude Code	✓	✗	✓	MCP	✓	✓

C. Benchmark Adaptation

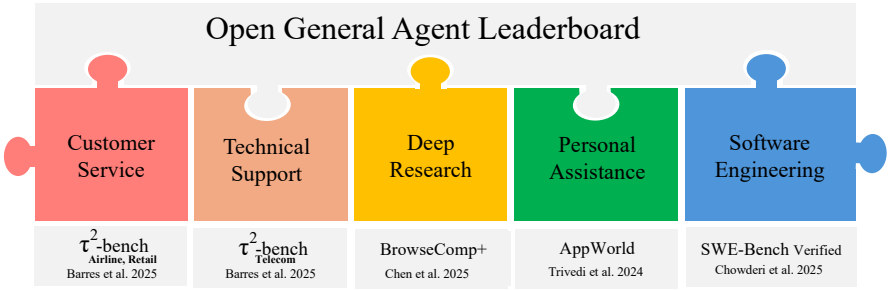

Figure 5. The six benchmarks evaluated in the Open General Agent Leaderboard, spanning software engineering, customer service, deep research, and personal assistance.

For each benchmark, this appendix shows a concrete task / context / actions example as it appears to the agent. The integration process itself — which reference agent we mirrored, what was kept vs. omitted, and what each task field encodes — is documented in Appendix A.1.1, and the released adaptor code lives at `src/exgentic/benchmarks/<benchmark>/` in the released repository.

C.1. SWE-Bench Verified Task Definition Example

TASK

```
Resolve the given issue by editing the repository files directly on a remote machine.

Repository directory on the remote machine: /testbed

## Issue to resolve:
Missing call `make_hashable` on `through_fields` in `ManyToManyRel`
Description

In 3.2 identity property has been added to all ForeignObjectRel to make it possible to
compare them.  A hash is derived from said identity and it's possible because identity
is a tuple.   To make limit_choices_to
hashable (one of this tuple elements), there's a call to make_hashable.
It happens that through_fields can be a list. In such case, this make_hashable call is
missing in ManyToManyRel.
For some reason it only fails on checking proxy model. I think proxy models have
29 checks and normal ones 24, hence the issue, but that's just a guess.

Minimal repro:
class Parent(models.Model):
    name = models.CharField(max_length=256)
class ProxyParent(Parent):
    class Meta:
        proxy = True
class Child(models.Model):
    parent = models.ForeignKey(Parent, on_delete=models.CASCADE)
    many_to_many_field = models.ManyToManyField(
        to=Parent,
        through="ManyToManyModel",
        through_fields=['child', 'parent'],
        related_name="something"
    )
class ManyToManyModel(models.Model):
    parent = models.ForeignKey(Parent, on_delete=models.CASCADE, related_name='+')
    child = models.ForeignKey(Child, on_delete=models.CASCADE, related_name='+')
    second_child = models.ForeignKey(Child, on_delete=models.CASCADE, null=True,
    default=None)

Which will result in:
    File "manage.py", line 23, in <module>
        main()
    File "manage.py", line 19, in main
        execute_from_command_line(sys.argv)

    ...

    File ".../django/db/models/fields/reverse_related.py", line 140, in __hash__
        return hash(self.identity)
TypeError: unhashable type: 'list'

Solution:
Add missing make_hashable call on self.through_fields in ManyToManyRel.

## Execution Environment & Access (STRICT):
All commands are executed on a remote machine that already contains the full
```

```
repository and all required system dependencies and prerequisites.

The remote machine is accessible ONLY via the `bash` action.
All interactions with the filesystem and environment, including reading files,
editing files, and running scripts, MUST be done through `bash`.

Each command is executed in a separate shell invocation; working directory changes
and environment variables do not persist between commands. Only files written to disk
persist. All bash commands are executed with the working directory set to /.

## Instructions:
You must fix the issue by directly modifying files in this repository.

ALL file edits MUST be performed using `bash`.
Explanations or code snippets in chat are not sufficient.

Only changes written to the actual repository files will be included in the final patch.

Modify only non-test source files in /testbed.

## Hard Boundaries:
- MODIFY ONLY: regular source code files in /testbed
- DO NOT MODIFY: tests or configuration files
- DO NOT use interactive editors
- DO NOT suggest changes without implementing them

## Patch & Submission Mechanics:
Submission captures the entire working tree using:
    git add -A && git diff --staged C0

This means:
- All intended fixes MUST appear in repository files
- Any file present at submission time WILL be included in the patch
- Temporary files must be removed before submission

## Recommended Workflow:
1. Analyze the codebase
2. Reproduce the issue
3. Edit using bash
4. Verify fix
5. Test edge cases

## Submission:
Use submit_patch exactly once with a short summary.

## Evaluation:
The patch will be applied to a hidden test suite and must pass all checks.
```

CONTEXT

Key	Value
(no entries)	

ACTIONS

- `bash(command: str)` — Run a bash command in the repo root and get the output.

- `finish(summary: str)` — Finish the task by submitting a brief summary; the system automatically computes the git patch from the repository changes.

C.2. BrowseComp+ Task Definition Example

TASK

```
Answer the provided question by performing search and document expansion as needed,
and submit your final answer.
Question: I need you to name the first and last name of the production controller
of a specific Indian film. The director of this film made one other movie the same year,
made his directorial debut in the 1980s, and has since directed over thirty films.
The film features the debut of an actor known for starring in a satirical
show influenced by real-world events, which is loosely adapted from a cartoon series
published in a famous magazine. This show has more than 4,000 episodes.
This actor has also played small roles in over 60 films. The film was released between
1990 and 2020, and a 1980s Hollywood movie inspired its storyline.
Additionally, an actor in the main cast is featured in a film with a title connected to
a  classic game of strategy, this film was released in the 2000s.

Note:
- The question has an answer discoverable through proper search.
- The question requires putting together information from different sources.

Your performance is scored based on:
 1. Most importantly, the correctness of the answer you assembled from different
 searches.
 2. Your effective use of search and your ability to retrieve all relevant
 information for the question.
 3. How efficiently you find all the relevant information, using as few searches
 as possible.

Important: During your work, Do NOT interact with the user or send any messages at any
point - messages will be ignored and are NOT considered a valid final answer.
The ONLY acceptable way to finish is by calling 'submit' with the required structured
fields.

Finish the session always by calling `submit`.
If you fail to find the answer, submit with exact_answer: "Can't find the answer.".
```

CONTEXT

Key	Value
(no entries)	

ACTIONS

- `search(query: str)` — Perform a search on the knowledge source; retrieves the top-k most relevant snippets, each trimmed to a documented token budget.
- `get_document(docid: str)` — Retrieve the full document by its document id (optional; enabled per benchmark configuration).
- `submit(exact_answer: str, explanation: str, confidence: float)` — Submit the final answer and end the session (finish action).

C.3. AppWorld Task Definition Example

TASK

```
Task from supervisor:
I have invited some of my friends to a reunion party via phone messages. I have made a
CSV to track who is coming or not in ~/documents/personal/ in my file system.
Please update RSVPs in it as per their latest replies.
```

CONTEXT

Key	Value
policy	This environment provides a set of applications, each exposing a predefined set of APIs that may be used to perform tasks on behalf of the supervisor. The applications include: api_docs, supervisor, amazon, phone, file_system, spotify, venmo, gmail, splitwise, simple_note, todoist. The available applications and their APIs are fixed for the task. Supervisor account credentials (such as emails, usernames, and passwords) are available through the supervisor application's APIs and are accessed from there when required. If an application requires an access token to perform authenticated operations, the access token is obtained by calling that application's authentication/login API using the credentials retrieved from the supervisor application. Access tokens are not provided by the supervisor application. References to people (e.g., friends, family, roommates) correspond to entries in the phone_contacts application. References to files or storage correspond to the file_system application, not the local machine filesystem. Time-based instructions (e.g., 'this month', 'yesterday') are interpreted with full calendar boundary ranges. If an API returns paginated results, all pages constitute the complete result. The environment consists only of the provided applications and their documented APIs and parameters. No additional endpoints, methods, arguments, or capabilities are assumed beyond those explicitly defined. When task execution is finished, the designated task-completion API is used to signal completion. If the task requires a final answer value, the answer is returned through that completion API. If the task cannot be completed using the available applications and APIs, the task may be marked as failed.
supervisor	{ "first_name": "Ashley", "last_name": "Moore", "email": "as_moore@gmail.com", "phone_number": "7336094411" }
datetime	2023-05-18T12:00:00

ACTIONS (∼468 TOTAL)

Generated programmatically from `world.task.api_docs.function_calling()`; one Unified Protocol action per AppWorld API across the nine apps, plus a benchmark-level `finish` mapped to `supervisor.complete_task`. Each action's name and signature mirror the AppWorld API; descriptions are the AppWorld-provided docstrings. Selected examples:

- `finish` — Mark the task as completed and (optionally) return a final answer; mapped from `supervisor.complete_task`.
- `supervisor.show_profile`
- `supervisor.show_addresses`
- `supervisor.show_payment_cards`
- `supervisor.show_account_passwords`
- `amazon.show_account`
- `amazon.signup`
- `amazon.delete_account`
- `amazon.update_account_name`
- `amazon.login`
- `amazon.logout`
- `amazon.clear_browsing_history`
- `amazon.search_sellers`
- `amazon.show_cart`
- `amazon.update_product_quantity_in_cart`

- `amazon.show_wish_list`
- `amazon.update_address`
- `amazon.show_product_reviews`
- `amazon.write_product_review`
- `amazon.show_product_questions`
- *(… 460+ additional tools omitted for brevity …)*
- `phone.search_contacts`
- `phone.send_text_message`
- `phone.show_alarm`
- `phone.update_alarm`
- `file_system.create_directory`
- `file_system.show_file`
- `spotify.show_account`
- `spotify.search_songs`
- `simple_note.create_note`
- `todoist.create_task`

C.4. τ^2-Bench Task Definition Example

TASK

```
You are a customer service agent that helps the user according to the <policy> provided
below.  Try to be helpful and always follow the policy.
```

CONTEXT

Key	Value
policy	# Airline Agent Policy The current time is 2024-05-15 15:00:00 EST. As an airline agent, you can help users **book**, **modify**, or **cancel** flight reservations. You also handle **refunds and compensation**. Before taking any actions that update the booking database (booking, modifying flights, editing baggage, changing cabin class, or updating passenger information), you must list the action details and obtain explicit user confirmation (yes) to proceed. You should not provide any information, knowledge, or procedures not provided by the user or available tools, or give subjective recommendations or comments. You should only make one tool call at a time, and if you make a tool call, you should not respond to the user simultaneously. If you respond to the user, you should not make a tool call at the same time. You should deny user requests that are against this policy. You should transfer the user to a human agent if and only if the request cannot be handled within the scope of your actions. To transfer, first make a tool call to transfer_to_human_agents, and then send the message 'YOU ARE BEING TRANSFERRED TO A HUMAN AGENT. PLEASE HOLD ON.' to the user. […] (full task text in released repository)

ACTIONS

The protocol `message` action carries replies to the simulated user; the remaining actions are translated from the τ^2-Bench domain's tool registry (one Unified Protocol action per registered tool, schema and description preserved). Listing for the airline subset:

- `message` — Send a message to the user.
- `book_reservation`
- `calculate`
- `cancel_reservation`
- `get_reservation_details`
- `get_user_details`
- `list_all_airports`
- `search_direct_flight`
- `search_onestop_flight`
- `send_certificate`
- `transfer_to_human_agents`
- `update_reservation_baggages`
- `update_reservation_flights`
- `update_reservation_passengers`
- `get_flight_status`

D. Detailed Benchmark Agent Interaction Example

This section demonstrates a complete interaction between a code-generation agent such as SmolAgents and the τ^2-Bench benchmark.

Agent Side. During initialization, the SmolAgent adaptor converts all Exgentic actions into lightweight Python wrapper functions. A standard SmolAgent instance is then created using the session's task definition and the set of wrapper functions.

When the agent invokes one of these wrapper functions, the wrapper places the corresponding action into an *action queue* and blocks while waiting for a response in an *observation queue*.

Later, when the orchestrator calls: $\text{action} = \text{CodeAgentWrapper.react}(\text{observation})$, the adaptor stores the observation in the *observation queue*, unblocking the agent-side wrapper function. The wrapper retrieves the observation and returns it to the agent as the result of the function call. Meanwhile, $\text{react}(\cdot)$ waits for the next action to appear in the *action queue*.

On the next invocation of a wrapper function, the agent places a new action in the *action queue*, which releases the blocked $\text{react}(\cdot)$ call. The action is then returned to the orchestrator, which forwards it to the benchmark session, obtains the next observation, and calls $\text{react}(\cdot)$ again.

This cycle continues until either the agent produces no further actions or the benchmark provides no further observations, signaling the end of the session.

Benchmark Side. During initialization in $\text{TauBenchBenchmark.start}()$, the list of available task names is retrieved from the τ^2-Bench codebase. When $\text{TauBenchBenchmark.next_session}()$ is invoked, a Session wrapper object is constructed. This wrapper defines the textual task description for the selected task and translates τ^2-Bench's OpenAI tool specifications

into Exgentic protocol actions. It then builds a proxy agent compatible with τ^2-Bench's internal agent API and begins executing τ^2-Bench code for the selected task.

When τ^2-Bench calls the proxy agent to obtain the next action given a simulated user message, the proxy agent stores the message in an *observation queue* and waits for an action to appear in the *action queue*. Once the orchestrator executes observation $= \mathrm{TauBenchBenchmark.step(action)}$, the benchmark wrapper stores the action in the *action queue*, allowing the proxy agent to resume and forward the action to τ^2-Bench. Meanwhile, $\mathrm{TauBenchBenchmark.step}(\cdot)$ blocks on the *observation queue*. When the proxy agent is called again by the τ^2-Bench code with the next simulated user message, it stores the message in the *observation queue*, enabling the observation to be returned to the orchestrator, which then passes it to the real agent.

E. Detailed Results

This appendix collects the full per-(agent, model, benchmark) results that back the headline claims in §4.1: the complete leaderboard, cost-efficiency for every (architecture, model) configuration, the generalist-vs-specialist Wilson-CI comparison, the tool-shortlisting ablation, average step counts split by success/failure, and per-model failure-step dynamics. Each subsection is self-contained: the same conclusions appear in compressed form in the main body, while here we give the raw numbers, sample sizes, and methodology footnotes needed to reproduce them.

Table 9

Table 9. Full agent-model configuration leaderboard (binary success rates per benchmark, deduped from runs.csv). All 25 (5 agents Œ 5 models) configurations across 6 benchmarks.

Agent	Model	App	BC+	SWE	T-Air	T-Ret	T-Tel	Mean
Solo	Opus	0.68	0.61	0.81	0.74	0.85	0.84	0.73
CC	Opus	0.66	0.53	0.74	0.66	0.83	0.76	0.67
Smol	Opus	0.70	0.61	0.65	0.72	0.78	0.58	0.66
React+Short	Gemini	0.55	0.48	0.71	0.70	0.82	0.73	0.62
React+Short	Opus	0.64	0.49	0.61	0.66	0.78	0.76	0.62
React	Gemini	0.50	0.48	0.71	0.70	0.82	0.73	0.61
React	Opus	0.61	0.49	0.61	0.66	0.78	0.76	0.61
Solo	Gemini	0.57	0.33	0.72	0.62	0.73	0.79	0.59
CC	Gemini	0.36	0.51	0.67	0.70	0.71	0.71	0.56
Smol	Gemini	0.13	0.57	0.76	0.68	0.75	0.88	0.56
React+Short	GPT	0.22	0.46	0.57	0.54	0.73	0.53	0.46
React	DeepSeek	0.09	0.36	0.69	0.56	0.82	0.71	0.46
React+Short	DeepSeek	0.04	0.36	0.69	0.56	0.82	0.71	0.45
React+Short	Kimi	0.10	0.34	0.57	0.62	0.65	0.83	0.43
React	Kimi	0.09	0.34	0.57	0.62	0.65	0.83	0.43
Smol	Kimi	0.11	0.33	0.58	0.56	0.72	0.71	0.42
CC	DeepSeek	0.03	0.48	0.64	0.28	0.65	0.61	0.42
Smol	DeepSeek	0.13	0.21	0.56	0.60	0.77	0.84	0.41
React	GPT	0.00	0.46	0.57	0.54	0.73	0.53	0.41
CC	GPT	0.00	0.43	0.58	0.48	0.64	0.55	0.39
Solo	GPT	0.00	0.48	0.55	0.50	0.53	0.53	0.39
Smol	GPT	0.07	0.26	0.53	0.60	0.68	0.71	0.38
Solo	DeepSeek	0.06	0.30	0.74	0.20	0.19	0.18	0.32
CC	Kimi	0.08	0.56	0.52	0.12	0.03	0.00	0.30
Solo	Kimi	0.08	0.35	0.57	0.00	0.01	0.00	0.25

presents the complete per-(agent, model, benchmark) leaderboard for all five models.

E.1. Cost-Efficiency per Configuration

Table 10 reports cost-efficiency (bench-weighted success divided by bench-weighted cost-per-task in dollars) for all 25 configurations, sorted from most to least efficient.

Tool-shortlisting cost effects (§4.1): adding shortlisting to ReAct reduces Claude Opus 4.5 cost from \$5.75 to \$3.78 per task ($-\$1.97$), while leaving GPT 5.2 approximately flat (\$0.17→\$0.26).

Table 10. Cost-efficiency for all 25 (architecture, model) configurations: bench-weighted success divided by bench-weighted cost per task (score/$). Sorted by efficiency, descending.

Configuration	Score	Cost/Task	Efficiency
ReAct + GPT 5.2	0.41	$0.17	2.43
OpenAI Solo + DeepSeek-V3.2	0.32	$0.13	2.43
OpenAI Solo + GPT 5.2	0.39	$0.19	2.03
ReAct Short + DeepSeek-V3.2	0.45	$0.23	1.95
Claude Code + DeepSeek-V3.2	0.42	$0.21	1.94
ReAct Short + GPT 5.2	0.46	$0.26	1.79
Smolagent + DeepSeek-V3.2	0.41	$0.23	1.76
ReAct + DeepSeek-V3.2	0.46	$0.38	1.21
OpenAI Solo + Kimi-K2.5	0.25	$0.22	1.12
ReAct Short + Kimi-K2.5	0.43	$0.41	1.05
Smolagent + GPT 5.2	0.38	$0.36	1.04
Claude Code + GPT 5.2	0.39	$0.38	1.04
ReAct Short + Gemini 3	0.62	$0.66	0.94
ReAct + Kimi-K2.5	0.43	$0.48	0.89
ReAct + Gemini 3	0.61	$0.81	0.76
Smolagent + Kimi-K2.5	0.42	$0.66	0.63
Claude Code + Kimi-K2.5	0.30	$0.61	0.50
Smolagent + Gemini 3	0.56	$1.85	0.30
Claude Code + Gemini 3	0.56	$2.47	0.23
OpenAI Solo + Gemini 3	0.59	$2.81	0.21
ReAct Short + Claude Opus 4.5	0.62	$3.78	0.16
Smolagent + Claude Opus 4.5	0.66	$4.39	0.15
ReAct + Claude Opus 4.5	0.61	$5.75	0.11
OpenAI Solo + Claude Opus 4.5	0.73	$8.54	0.09
Claude Code + Claude Opus 4.5	0.67	$8.03	0.08

E.2. References to Leaderboards

For reference, SWE-Bench Verified leaderboard top reported domain-specific agent achieves 0.79[6], BrowseComp+ and AppWorld are 0.80[7], and 0.73[8], respectively. τ^2-Bench Airline (0.73), Retail (0.86), and Telecom (0.98)[9].

E.3. Generalist vs. Specialist Wilson CI Comparison

Table 11 reports the per-benchmark Wilson 95% half-width and the gap between our best generalist configuration and the top reported specialist score (§4.1, App. E.2). The *Indistinguishable* verdict applies when the gap is within the per-benchmark Wilson half-width (or favors the generalist); on BrowseComp+ (gap 19pp) and τ^2-Bench-Telecom (gap 10pp) the specialist leads beyond Wilson uncertainty.

Table 11. Per-benchmark Wilson 95% half-widths and generalist–specialist gaps. n is the number of evaluated instances. "HW gen/spec" are Wilson half-widths in percentage points at the respective scores. "Gap" is specialist minus generalist (positive favors specialist). The four "Indistinguishable" rows back the body claim of parity on 4 of 6 benchmarks.

Benchmark	Best Generalist	n	Gen	Spec	Gap (pp)	HW gen/spec (pp)	Verdict
SWE-Bench Verified	OpenAI Solo+Claude Opus 4.5	100	0.81	0.79	-2	7.5 / 8.0	Indistinguishable
BrowseComp+	Smolagent+Claude Opus 4.5	100	0.61	0.80	$+19$	9.5 / 8.0	Specialist leads
τ^2-Bench-Airline	OpenAI Solo+Claude Opus 4.5	50	0.74	0.73	-1	12.0 / 12.5	Indistinguishable
τ^2-Bench-Retail	OpenAI Solo+Claude Opus 4.5	100	0.85	0.86	$+1$	7.0 / 7.0	Indistinguishable
τ^2-Bench-Telecom	Smolagent+Gemini 3	100	0.88	0.98	$+10$	6.5 / 3.5	Specialist leads
AppWorld	Smolagent+Claude Opus 4.5	100	0.70	0.73	$+3$	9.0 / 8.5	Indistinguishable

[6] SWE-Bench Verified
[7] BrowseComp+
[8] AppWorld
[9] τ^2-Bench

E.4. Tool Shortlisting Ablation

Table 12 isolates the per-cell impact of adding tool shortlisting to a vanilla ReAct architecture, by model. AppWorld is the only tool-rich benchmark in the suite ($\sim$468 actions); shortlisting changes nothing on the others (whose action sets fit within all five LLMs' tool limits) so the bench-weighted column reduces to ¼ of the AppWorld delta. The largest gain is on GPT 5.2 ($+22$pp on AppWorld, $+5$pp aggregate), where the 128-tool API limit otherwise drives the cell to 0.00. DeepSeek-V3.2 is the lone regression (-5pp on AppWorld, -1pp aggregate), indicating shortlisting is broadly but not universally helpful (cf. §4.1).

Table 12. Tool-shortlisting ablation: success-rate delta from adding shortlisting to a vanilla ReAct agent (ReAct+ReAct Short minus ReAct), in percentage points. Aggregate is bench-weighted across all six benchmarks.

Model	**AppWorld Δ (pp)**	**Aggregate Δ (pp)**	**Cost Δ/task**
Claude Opus 4.5	$+3$	$+0.7$	$-\$1.97$
Gemini 3	$+5$	$+1.2$	$-\$0.15$
GPT 5.2	$+22$	$+5.5$	$+\$0.09$
DeepSeek-V3.2	-5	-1.3	$-\$0.15$
Kimi-K2.5	$+1$	$+0.3$	$-\$0.07$

E.5. Step Counts

Table 13. Average steps per benchmark and architecture, split by successful vs. failed sessions. The three closed-source backbones (Claude Opus 4.5, Gemini 3, GPT 5.2) are aggregated within each (benchmark, architecture) cell. Zero-step sessions are excluded (4.0% of closed-source runs); step counts are capped at 50 to limit outlier influence (7.1% of remaining runs). SWE-Bench Verified success is patch-pass ($\texttt{score} \geq 1$); other benchmarks use binary session success. Per-model raw values for all five backbones are in App. E.6.

Benchmark	**Claude Code Succ**	**Claude Code Fail**	**OpenAI Solo Succ**	**OpenAI Solo Fail**	**Smolagent Succ**	**Smolagent Fail**	**ReAct Succ**	**ReAct Fail**	**ReAct Short Succ**	**ReAct Short Fail**
AppWorld	23.67	38.56	26.41	39.39	25.69	34.17	13.24	27.91	12.34	21.41
BrowseComp+	13.90	23.64	15.23	17.96	15.89	23.83	9.28	15.53	9.28	15.53
SWE-Bench Verified	27.69	32.21	27.76	29.30	29.28	32.03	28.38	34.22	28.38	34.22
airline	10.43	13.02	10.19	13.65	10.39	14.06	9.31	12.18	9.31	12.18
retail	11.73	11.54	11.36	9.81	11.48	11.31	10.81	11.52	10.81	11.52
telecom	12.98	12.25	13.06	13.26	11.99	12.75	13.23	15.84	13.23	15.84
weighted_avg	**19.24**	**26.67**	**20.24**	**24.72**	**20.54**	**25.68**	**15.50**	**22.71**	**15.28**	**21.08**

Table 14. Percentage difference in interactions between failed and successful runs (from the absolute counts above). Positive values mean failures take more interactions; negative values mean they take fewer. The Average row is the bench-weighted simple mean of per-benchmark percentages (1/4 each non-tau, 1/12 each tau subdomain).

Benchmark	**Claude Code**	**OpenAI Solo**	**Smolagent**	**ReAct**	**ReAct Short**
AppWorld	63%	49%	33%	111%	74%
BrowseComp+	70%	18%	50%	67%	67%
SWE-Bench Verified	16%	6%	9%	21%	21%
τ^2-Bench-Airline	25%	34%	35%	31%	31%
τ^2-Bench-Retail	-2%	-14%	-2%	7%	7%
τ^2-Bench-Telecom	-6%	2%	6%	20%	20%
Average	39%	20%	26%	54%	45%

E.6. Per-Model Failure-Step Dynamics

Table 15 reports the same percentage-difference computation as Table 14 but un-pooled across backbone models, so the open-weight rows are directly visible. The closed-source rows reproduce the body's bench-weighted averages within rounding when reaggregated across (Claude Opus 4.5, Gemini 3, GPT 5.2). The two open-weight backbones depart from the closed-source pattern most sharply on autonomous architectures (OpenAI Solo, Claude Code): DeepSeek-V3.2+OpenAI Solo reaches -27% (failures shorter than success), driven by the protocol-violation early-termination dynamic characterized in §4.7; Kimi-K2.5+OpenAI Solo and Kimi-K2.5+Claude Code have insufficient τ^2-Bench data after the same filter for some cells ("–"). Methodology is identical to Table 13: zero-step sessions excluded, step counts capped at 50,

SWE-Bench Verified success is patch-pass. Reproduction script: `tools/audit_failure_steps.py` in the released repository.

Table 15. Per-(model, agent) bench-weighted percentage difference in steps between failed and successful runs. Positive values mean failures take more steps; negative values mean failures terminate earlier than success. AppWorld, BrowseComp+, and SWE-Bench Verified contribute weight 1/4 each; the three τ^2-Bench subdomains contribute weight 1/12 each. "–" indicates a cell with insufficient succ/fail data after filtering.

Model	Agent	App	BC+	SWE	T-Air	T-Ret	T-Tel	Bench-wgt
Claude Opus 4.5	ReAct	+127%	+115%	+41%	+33%	+10%	+63%	+79.8%
Claude Opus 4.5	ReAct Short	+163%	+115%	+41%	+33%	+10%	+63%	+88.7%
Claude Opus 4.5	Smolagent	+40%	+97%	+26%	+32%	−1%	−12%	+42.4%
Claude Opus 4.5	OpenAI Solo	+71%	+86%	+23%	+32%	−9%	+38%	+50.1%
Claude Opus 4.5	Claude Code	+93%	+56%	+29%	+28%	+8%	+43%	+51.1%
Gemini 3	ReAct	+98%	+38%	+4%	+26%	+4%	+22%	+39.3%
Gemini 3	ReAct Short	+136%	+38%	+4%	+26%	+4%	+22%	+49.0%
Gemini 3	Smolagent	+76%	+151%	+30%	+53%	+2%	+35%	+71.6%
Gemini 3	OpenAI Solo	+33%	+19%	+14%	+20%	−9%	+24%	+19.4%
Gemini 3	Claude Code	+58%	+154%	+23%	+41%	+3%	−40%	+59.2%
GPT 5.2	ReAct	–	+16%	+20%	+31%	+8%	+11%	+17.6%
GPT 5.2	ReAct Short	+11%	+16%	+20%	+31%	+8%	+11%	+15.9%
GPT 5.2	Smolagent	+91%	−23%	−3%	+27%	−3%	+15%	+19.8%
GPT 5.2	OpenAI Solo	–	+4%	+20%	+63%	−7%	−1%	+13.9%
GPT 5.2	Claude Code	–	+14%	+7%	+20%	−9%	−4%	+7.5%
DeepSeek-V3.2	ReAct	+54%	+43%	−2%	+30%	+13%	+70%	+33.1%
DeepSeek-V3.2	ReAct Short	+111%	+43%	−2%	+30%	+13%	+70%	+47.3%
DeepSeek-V3.2	Smolagent	+52%	−26%	+6%	+47%	+8%	+54%	+17.2%
DeepSeek-V3.2	OpenAI Solo	−19%	−34%	−37%	+25%	−47%	−33%	−27.1%
DeepSeek-V3.2	Claude Code	+92%	+68%	−8%	+38%	+0%	+29%	+43.7%
Kimi-K2.5	ReAct	+192%	+58%	+0%	+27%	+11%	+20%	+67.6%
Kimi-K2.5	ReAct Short	+33%	+58%	+0%	+27%	+11%	+20%	+27.7%
Kimi-K2.5	Smolagent	+42%	+62%	+9%	+75%	+1%	+28%	+36.7%
Kimi-K2.5	OpenAI Solo	+71%	+76%	−0%	–	−61%	–	+37.8%
Kimi-K2.5	Claude Code	+37%	+131%	+2%	−5%	−35%	–	+43.0%

F. Statistical Significance

We assess the statistical significance of the benchmark results using a coordinated battery of tests matched to each axis of comparison: paired McNemar tests on binary success outcomes for shared (benchmark, task) pairs, paired t-tests on continuous scores for cross-architecture and cross-model contrasts, Wilson 95% confidence intervals for per-benchmark success rates, and two-proportion z-tests for protocol-violation rate comparisons. Multiple-testing is controlled with Benjamini–Hochberg and Benjamini–Yekutieli corrections at $\alpha = 0.05$; all qualitative conclusions in this appendix and §4.1 survive both. The evaluation consists of six benchmark configurations with the following instance counts: AppWorld (100), BrowseComp+ (100), SWE-Bench Verified (100), τ^2-Bench Airline (50), τ^2-Bench Retail (100), and τ^2-Bench Telecom (100), for a total of 550 instances per agent-model configuration — an aggregation level large enough that cross-model differences of even a few percentage points reach $p < 0.0001$, while within-benchmark single-cell contrasts at $n = 100$ remain noisy and are reported with their Wilson half-widths throughout.

For a single benchmark with n = 100 binary trials, the 95% Wilson (Wilson, 1927) confidence-interval half-width typically ranges from 7 to 9.5 percentage points when the observed success rate lies between 0.3 and 0.8, the region where most leading agent configurations perform. This means that differences smaller than approximately 8–10 percentage points on individual benchmarks should be interpreted cautiously, as they fall within normal statistical uncertainty. To obtain a more stable measure, we compute a weighted aggregate score across all benchmark instances. Under the assumption

that benchmarks are independent of one another, this yields an effective sample size of $n = 550$ per agent-model configuration. The corresponding 95% delta-method confidence-interval half-width for the aggregated score is substantially smaller, typically in the range of 4–5 percentage points. The paper's central claims operate at even larger aggregation levels: backbone-model contrasts aggregate across 5 architectures $\times$ 550 instances $= 2{,}750$ observations per model (giving a 95% CI half-width of approximately ± 1.8pp), and the η^2 variance decomposition in Section 4.3 uses all $\sim 8{,}250$ observations across 15 closed-source agent configurations. At these sample sizes, the model-axis Claude Opus 4.5-vs-GPT 5.2 gap (26pp, aggregated across architectures and benchmarks) and similar cross-model differences are highly significant, even though within-benchmark comparisons at $n = 100$ remain noisy. These levels of statistical uncertainty are standard across existing agentic leaderboards: most widely used agent-evaluation platforms report confidence intervals on the order of only a few percentage points, reflecting the inherent variability of evaluations on datasets of similar size.

We use two paired tests on shared (benchmark, task) outcomes to enhance statistical power. A pooled McNemar test on binary success outcomes compares each configuration against the top-ranked configuration; this is the basis for the leaderboard ranking claims in the body. The paired t-test reported in the tables below operates on continuous scores and isolates per-model agent-architecture contrasts. For the model-axis ranking in §4.1, we run pairwise paired tests on continuous scores aligned across (benchmark, task, agent) tuples between each pair of models; all pairs differ at $p < 0.0001$. For the agent-axis ranking we run the same paired tests on (benchmark, task, model) tuples between each pair of agent architectures; no pair reaches $p < 0.1$.

We apply Benjamini–Hochberg and Benjamini–Yekutieli corrections at $\alpha = 0.05$. All qualitative conclusions in this appendix and the §4.1 leaderboard ranking claims (cross-model paired-t separation $p < 0.0001$ between any two model tiers; within-model architecture pairs largely $p > 0.1$) survive both corrections. Raw p-values are shown in the tables for transparency.

F.1. Within-Model Scaffold Sensitivity

Table 16 reports, for each model, the best vs. worst agent architecture by bench-weighted mean success and the paired t-test on shared (benchmark, task) pairs. This is the basis for our claim that agent-architecture choice has small effect on closed-source models (7–12pp swing) but largely determines open-weight performance (14–18pp swing).

Table 16. Within-model agent-architecture sensitivity. For each model we report the best and worst agent architecture by bench-weighted mean success, plus a paired t-test on shared (benchmark, task) pairs comparing the two. Closed-source models swing 7–12pp; open-weight models swing 14–18pp with highly significant best-vs-worst tests.

Model	Best arch.	Mean	Worst arch.	Mean	Δ (pp)	n	p
Opus	Solo	0.727	React	0.610	+11.7	105	0.100
Gemini	React+Short	0.622	Smol	0.557	+6.6	5	0.317
GPT	React+Short	0.463	Smol	0.380	+8.3	105	0.005
DeepSeek	React	0.459	Solo	0.322	+13.7	550	<0.001
Kimi	React+Short	0.428	Solo	0.250	+17.7	547	<0.001

F.2. Pairwise Scaffold Comparisons

For each model, we report all pairwise agent-architecture p-values from a paired t-test on shared (benchmark, task) pairs (Tables 17–21).

Table 17. Pairwise agent-architecture paired-t-test p-values for Opus. Lower triangle: p; upper triangle: Δ (row $-$ column, in pp). $***p < 0.001$, $**\ p < 0.01$, $*p < 0.05$.

	React	React+Short	Smol	Solo	CC
React	—	+0.0	-20.0	-7.6	-20.0
React+Short	<0.001***	—	+0.0	-20.0	-16.7
Smol	0.317	1.000	—	+0.0	-16.7
Solo	0.100	0.317	<0.001***	—	+0.0
CC	0.317	0.317	0.317	<0.001***	—

Table 18. Pairwise agent-architecture paired-t-test p-values for Gemini. Lower triangle: p; upper triangle: Δ (row $-$ column, in pp). $***p < 0.001$, $**\ p < 0.01$, $*p < 0.05$.

	React	React+Short	Smol	Solo	CC
React	—	-4.8	+20.0	-5.7	-20.0
React+Short	0.224	—	+20.0	-1.0	-20.0
Smol	0.317	0.317	—	+0.0	-33.3
Solo	0.256	0.836	1.000	—	-40.0
CC	0.317	0.317	0.114	0.102	—

Table 19. Pairwise agent-architecture paired-t-test p-values for GPT. Lower triangle: p; upper triangle: Δ (row $-$ column, in pp). $***p < 0.001$, $**\ p < 0.01$, $*p < 0.05$.

	React	React+Short	Smol	Solo	CC
React	—	-21.0	-7.6	+0.0	-1.9
React+Short	<0.001***	—	+13.3	+21.0	+19.0
Smol	0.003**	0.005**	—	+7.6	+5.7
Solo	1.000	<0.001***	0.003**	—	-1.9
CC	0.155	<0.001***	0.031*	0.155	—

Table 20. Pairwise agent-architecture paired-t-test p-values for DeepSeek. Lower triangle: p; upper triangle: Δ (row $-$ column, in pp). $***p < 0.001$, $**\ p < 0.01$, $*p < 0.05$.

	React	React+Short	Smol	Solo	CC
React	—	+0.9	-0.5	+36.0	+7.1
React+Short	0.025*	—	-1.5	+35.1	+6.2
Smol	0.768	0.442	—	+36.5	+7.6
Solo	<0.001***	<0.001***	<0.001***	—	-28.9
CC	<0.001***	0.003**	<0.001***	<0.001***	—

Table 21. Pairwise agent-architecture paired-t-test p-values for Kimi. Lower triangle: p; upper triangle: Δ (row $-$ column, in pp). $***p < 0.001$, $**\ p < 0.01$, $*p < 0.05$.

	React	React+Short	Smol	Solo	CC
React	—	-0.2	+1.5	+34.2	+29.5
React+Short	0.564	—	+1.7	+34.4	+29.7
Smol	0.442	0.389	—	+33.0	+28.2
Solo	<0.001***	<0.001***	<0.001***	—	-4.6
CC	<0.001***	<0.001***	<0.001***	0.004**	—

F.3. Variance Decomposition (Closed-Source)

Table 22 reproduces the variance decomposition reported in §4.3 using two methodologies that both match the Exgentic `analyse model-agent` CLI.

Two-way ANOVA on cell-level success scores for the 15 closed-source configurations yields model main effect $F(2, 78) = 34.95$, $p < 10^{-10}$; agent main effect $F(4, 78) = 0.16$, $p = 0.96$; benchmark $F(5, 78) = 25.20$, $p < 10^{-14}$. The agent main effect emerges only when open-weight models are included ($F(4, 136) = 3.82$, $p = 0.006$), reflecting their architectural sensitivity (§4.7).

Table 22. Variance decomposition (closed-only, 90 deduped (agent, model, bench) cells). Both methodologies match the Exgentic `analyse model-agent` CLI exactly. Axis decomposition (model 27.8% vs. agent 0.5%) is the headline ratio cited in §4.3; additive decomposition over (model, architecture) cell means yields a 5.4% interaction term.

Component	Axis decomp. (%)	Additive on cells (%)
Model	27.8	93.9
Agent	0.5	0.7
Benchmark	40.7	–
Interaction	–	5.4
Residual (within-cell + interactions)	31.0	–
Model / Agent ratio	57.8×	128.6×

F.4. Generality Sinks Decomposition

Per-benchmark decomposition of the open-weight gap to closed-source: a *capability gap* (closed-best cell minus open-best cell) and an *architectural-sensitivity gap* (open-weight per-model architectural spread minus closed-source per-model spread, averaged across models). 95% bootstrap CIs (2,000 resamples, parametric Binomial bootstrap on per-cell success counts) in brackets.

Benchmark	**Capability gap (pp)**	**Arch.-sens. gap (pp)**	**Sink type**
AppWorld	$+57.0$ $[+50, +66]$	-18.5 $[-25, -11]$	Benchmark sink
SWE-Bench Verified	$+7.0$ $[-2, +16]$	$+0.2$ $[-9, +9]$	(none)
BrowseComp+	$+5.0$ $[-3, +19]$	$+5.8$ $[-6, +16]$	(none)
τ^2-Bench-Airline	$+12.0$ $[+3, +20]$	$+41.7$ $[+30, +47]$	Architecture sink
τ^2-Bench-Retail	$+3.0$ $[-5, +10]$	$+54.6$ $[+44, +60]$	Architecture sink
τ^2-Bench-Telecom	$+4.0$ $[-5, +10]$	$+54.2$ $[+44, +61]$	Architecture sink

F.5. Step-Zero Rates

Two-proportion z tests on τ^2-Bench autonomous (Claude Code + OpenAI Solo) sessions, $n = 500$ for each open-weight model and $n = 1{,}500$ for closed-source: Kimi-K2.5 (94%) vs closed-source (1.7%), $z = +41.4$; DeepSeek-V3.2 (31%) vs closed-source (1.7%), $z = +20.0$; Kimi-K2.5 vs DeepSeek-V3.2, $z = +20.5$. All pairwise $p < 10^{-15}$.

F.6. Cross-Benchmark Spearman Correlations

Table 23 reports Spearman rank correlations between benchmarks across the 15 closed-model configurations.

Table 23. Cross-benchmark Spearman rank correlations across the 15 closed-model configurations. Predominantly positive correlations indicate that model identity drives consistency across benchmarks.

	App	BC+	SWE	T-Air	T-Ret	T-Tel
App	—	+0.67	+0.60	+0.73	+0.81	+0.57
BC+	+0.67	—	+0.62	+0.75	+0.60	+0.44
SWE	+0.60	+0.62	—	+0.69	+0.73	+0.80
T-Air	+0.73	+0.75	+0.69	—	+0.73	+0.56
T-Ret	+0.81	+0.60	+0.73	+0.73	—	+0.66
T-Tel	+0.57	+0.44	+0.80	+0.56	+0.66	—

F.7. Full Leaderboard with Open-Weight Models

Table 24 extends the headline leaderboard with all 25 (5 agents × 5 models) configurations.

Table 24. Full agent-configuration leaderboard: per-(agent architecture, backbone model, benchmark) success rates (binary, deduped from runs.csv via the same path the Exgentic CLI uses) for all 25 agent configurations. Rightmost column is the bench-weighted mean (1/4 each non-tau2, 1/12 each tau2 subdomain).

Agent	Model	App	BC+	SWE	T-Air	T-Ret	T-Tel	Mean
React	Opus	0.61	0.49	0.61	0.66	0.78	0.76	0.610
React+Short	Opus	0.64	0.49	0.61	0.66	0.78	0.76	0.617
Smol	Opus	0.70	0.61	0.65	0.72	0.78	0.58	0.663
Solo	Opus	0.68	0.61	0.81	0.74	0.85	0.84	0.727
CC	Opus	0.66	0.53	0.74	0.66	0.83	0.76	0.670
React	Gemini	0.50	0.48	0.71	0.70	0.82	0.73	0.610
React+Short	Gemini	0.55	0.48	0.71	0.70	0.82	0.73	0.622
Smol	Gemini	0.13	0.57	0.76	0.68	0.75	0.88	0.557
Solo	Gemini	0.57	0.33	0.72	0.62	0.73	0.79	0.585
CC	Gemini	0.36	0.51	0.67	0.70	0.71	0.71	0.562
React	GPT	0.00	0.46	0.57	0.54	0.73	0.53	0.408
React+Short	GPT	0.22	0.46	0.57	0.54	0.73	0.53	0.463
Smol	GPT	0.07	0.26	0.53	0.60	0.68	0.71	0.380
Solo	GPT	0.00	0.48	0.55	0.50	0.53	0.53	0.386
CC	GPT	0.00	0.43	0.58	0.48	0.64	0.55	0.392
React	DeepSeek	0.09	0.36	0.69	0.56	0.82	0.71	0.459
React+Short	DeepSeek	0.04	0.36	0.69	0.56	0.82	0.71	0.446
Smol	DeepSeek	0.13	0.21	0.56	0.60	0.77	0.84	0.409
Solo	DeepSeek	0.06	0.30	0.74	0.20	0.19	0.18	0.322
CC	DeepSeek	0.03	0.48	0.64	0.28	0.65	0.61	0.416
React	Kimi	0.09	0.34	0.57	0.62	0.65	0.83	0.425
React+Short	Kimi	0.10	0.34	0.57	0.62	0.65	0.83	0.428
Smol	Kimi	0.11	0.33	0.58	0.56	0.72	0.71	0.420
Solo	Kimi	0.08	0.35	0.57	0.00	0.01	0.00	0.250
CC	Kimi	0.08	0.56	0.52	0.12	0.03	0.00	0.303

F.8. Data Quality and Run Provenance

Table 25 lists, per cell, the canonical run's completed/planned task counts and the total number of run rows present in `runs.csv`.

Table 25. Data quality per cell. "Completed/Planned" is the canonical (most-completed) run; "#runs" is the number of run rows for that cell in raw `runs.csv` (a value above 1 indicates re-runs or partial-run artifacts that the dedup step collapses to a single row).

Agent	Model	Benchmark	Completed	Planned	#runs
React	Opus	App	100	100	1
React	Opus	BC+	0	100	1
React	Opus	SWE	99	99	1
React	Opus	T-Air	50	50	1
React	Opus	T-Ret	100	100	1
React	Opus	T-Tel	100	100	1
React+Short	Opus	App	100	100	1
React+Short	Opus	BC+	0	100	1
React+Short	Opus	SWE	99	99	1
React+Short	Opus	T-Air	50	50	1
React+Short	Opus	T-Ret	100	100	1
React+Short	Opus	T-Tel	100	100	1
Smol	Opus	App	100	100	1
Smol	Opus	BC+	100	100	1
Smol	Opus	SWE	100	100	1
Smol	Opus	T-Air	50	50	1
Smol	Opus	T-Ret	100	100	1
Smol	Opus	T-Tel	100	100	1
Solo	Opus	App	100	100	1
Solo	Opus	BC+	100	100	1
Solo	Opus	SWE	83	83	1
Solo	Opus	T-Air	50	50	1
Solo	Opus	T-Ret	100	100	1
Solo	Opus	T-Tel	100	100	1
CC	Opus	App	100	100	1
CC	Opus	BC+	0	51	1
CC	Opus	SWE	97	97	1
CC	Opus	T-Air	50	50	1
CC	Opus	T-Ret	100	100	1
CC	Opus	T-Tel	100	100	1
React	Gemini	App	99	100	1
React	Gemini	BC+	0	100	1

Continued on next page

(Table 25 continued)

Agent	Model	Benchmark	Completed	Planned	#runs
React	Gemini	SWE	100	100	1
React	Gemini	T-Air	50	50	1
React	Gemini	T-Ret	100	100	1
React	Gemini	T-Tel	100	100	1
React+Short	Gemini	App	100	100	1
React+Short	Gemini	BC+	0	100	1
React+Short	Gemini	SWE	100	100	1
React+Short	Gemini	T-Air	50	50	1
React+Short	Gemini	T-Ret	100	100	1
React+Short	Gemini	T-Tel	100	100	1
Smol	Gemini	App	100	100	1
Smol	Gemini	BC+	0	100	1
Smol	Gemini	SWE	99	99	1
Smol	Gemini	T-Air	50	50	1
Smol	Gemini	T-Ret	100	100	1
Smol	Gemini	T-Tel	100	100	1
Solo	Gemini	App	98	100	1
Solo	Gemini	BC+	0	99	1
Solo	Gemini	SWE	94	94	1
Solo	Gemini	T-Air	50	50	1
Solo	Gemini	T-Ret	100	100	1
Solo	Gemini	T-Tel	89	100	1
CC	Gemini	App	100	100	1
CC	Gemini	BC+	0	100	1
CC	Gemini	SWE	100	100	1
CC	Gemini	T-Air	50	50	1
CC	Gemini	T-Ret	100	100	1
CC	Gemini	T-Tel	100	100	1
React	GPT	App	0	100	1
React	GPT	BC+	0	100	1
React	GPT	SWE	100	100	1
React	GPT	T-Air	50	50	1
React	GPT	T-Ret	100	100	1
React	GPT	T-Tel	100	100	1
React+Short	GPT	App	100	100	1
React+Short	GPT	BC+	0	100	1
React+Short	GPT	SWE	100	100	1
React+Short	GPT	T-Air	50	50	1
React+Short	GPT	T-Ret	100	100	1
React+Short	GPT	T-Tel	100	100	1
Smol	GPT	App	98	100	1
Smol	GPT	BC+	0	100	1
Smol	GPT	SWE	99	99	1
Smol	GPT	T-Air	50	50	1
Smol	GPT	T-Ret	100	100	1
Smol	GPT	T-Tel	100	100	1
Solo	GPT	App	0	100	1
Solo	GPT	BC+	100	100	1
Solo	GPT	SWE	99	99	1
Solo	GPT	T-Air	50	50	1
Solo	GPT	T-Ret	100	100	1
Solo	GPT	T-Tel	100	100	1
CC	GPT	App	0	100	1
CC	GPT	BC+	100	100	1
CC	GPT	SWE	100	100	1
CC	GPT	T-Air	50	50	1
CC	GPT	T-Ret	100	100	1
CC	GPT	T-Tel	100	100	1
React	DeepSeek	App	99	100	1
React	DeepSeek	BC+	99	100	1
React	DeepSeek	SWE	96	100	1
React	DeepSeek	T-Air	50	50	1
React	DeepSeek	T-Ret	100	100	1
React	DeepSeek	T-Tel	100	100	1
React+Short	DeepSeek	App	100	100	1
React+Short	DeepSeek	BC+	99	100	1
React+Short	DeepSeek	SWE	96	100	1
React+Short	DeepSeek	T-Air	50	50	1
React+Short	DeepSeek	T-Ret	100	100	1
React+Short	DeepSeek	T-Tel	100	100	1
Smol	DeepSeek	App	100	100	1
Smol	DeepSeek	BC+	100	100	1
Smol	DeepSeek	SWE	100	100	1
Smol	DeepSeek	T-Air	50	50	1
Smol	DeepSeek	T-Ret	98	100	1
Smol	DeepSeek	T-Tel	100	100	1
Solo	DeepSeek	App	48	100	1

Continued on next page

(Table 25 continued)

Agent	Model	Benchmark	Completed	Planned	#runs
Solo	DeepSeek	BC+	62	100	1
Solo	DeepSeek	SWE	38	100	1
Solo	DeepSeek	T-Air	50	50	1
Solo	DeepSeek	T-Ret	100	100	1
Solo	DeepSeek	T-Tel	100	100	1
CC	DeepSeek	App	100	100	1
CC	DeepSeek	BC+	100	100	1
CC	DeepSeek	SWE	100	100	1
CC	DeepSeek	T-Air	50	50	1
CC	DeepSeek	T-Ret	100	100	1
CC	DeepSeek	T-Tel	100	100	1
React	Kimi	App	99	100	1
React	Kimi	BC+	98	100	1
React	Kimi	SWE	98	100	1
React	Kimi	T-Air	50	50	1
React	Kimi	T-Ret	99	100	1
React	Kimi	T-Tel	100	100	1
React+Short	Kimi	App	96	100	1
React+Short	Kimi	BC+	98	100	1
React+Short	Kimi	SWE	98	100	1
React+Short	Kimi	T-Air	50	50	1
React+Short	Kimi	T-Ret	99	100	1
React+Short	Kimi	T-Tel	100	100	1
Smol	Kimi	App	100	100	1
Smol	Kimi	BC+	100	100	1
Smol	Kimi	SWE	92	100	1
Smol	Kimi	T-Air	50	50	1
Smol	Kimi	T-Ret	98	100	1
Smol	Kimi	T-Tel	99	100	1
Solo	Kimi	App	52	100	1
Solo	Kimi	BC+	62	100	1
Solo	Kimi	SWE	97	100	1
Solo	Kimi	T-Air	50	50	1
Solo	Kimi	T-Ret	100	100	1
Solo	Kimi	T-Tel	100	100	1
CC	Kimi	App	100	100	1
CC	Kimi	BC+	100	100	1
CC	Kimi	SWE	98	100	1
CC	Kimi	T-Air	50	50	1
CC	Kimi	T-Ret	100	100	1
CC	Kimi	T-Tel	100	100	1

G. Reproducibility Details

Appendix G provides everything required to reproduce the leaderboard, the statistical analyses, and the auto-generated appendix tables: the shared run configuration (§G.1), provider-default inference settings (Table 27), the audit scripts that recompute every numeric claim from the raw session log (§G.3), the data layout and key methodology conventions (§G.4), and the exact `task_id` pool evaluated in every cell (§G.5). The framework code, the benchmark adaptors, the per-session results file (`sessions.csv`), and the per-run aggregated file (`runs.csv`) are released alongside the paper.

G.1. Shared Run Configuration

Table 26 summarizes the configuration shared by all runs. The full per-run configuration (agent version strings, system prompts, tool description templates, benchmark adaptor code) is released with the framework; agent versions and prompts are documented in Appendix B, per-benchmark adaptations in Appendix C.

Parameter	Value
LLMs	GPT 5.2, Claude Opus 4.5, Gemini 3 Pro, DeepSeek-V3.2, Kimi-K2.5
LLM sampling	Provider API defaults (temperature, top-p)
Reasoning mode	Provider default for each model (not manually toggled)
Max turns per task	100
Agent architectures	ReAct, ReAct Short, Smolagent, OpenAI Solo, Claude Code
Benchmarks	AppWorld, BrowseComp+, SWE-Bench Verified, τ^2-Bench (3 subdomains)
Instances per benchmark	100 (τ^2-Airline: 50); 550 total per agent-model
Total configurations	150 (5 agents $\times$ 5 models $\times$ 6 benchmarks)
Total evaluation cost	$\sim$\$20K

Table 26. Configuration shared across all runs. Per-benchmark prompts and tool description templates follow the protocol in Appendix C.

G.2. Inference Arguments

Provider defaults. For every $(\text{agent}, \text{model}, \text{benchmark})$ cell we use the provider's documented default for every sampling and reasoning parameter. Table 27 reports those defaults verbatim from each provider's official documentation. We do not re-tune sampling per agent, model, or benchmark, to avoid confounding the model-versus-agent comparison with hyperparameter search; the cost is that absolute scores under-estimate what a tuned configuration could achieve.

Model	Endpoint	temp.	top-p	top-k	thinking
Claude Opus 4.5	AWS Bedrock	1.0	undisclosed	undisclosed	off
GPT 5.2	Azure OpenAI	unsupported	unsupported	unsupported	off
Gemini 3 Pro	GCP Vertex AI	1.0	0.95	64	high
DeepSeek-V3.2	Azure / Foundry	1.0	1.0	unsupported	enabled
Kimi-K2.5	Azure / Foundry	1.0*	1.0	unsupported	enabled

Table 27. Provider-documented default inference settings, used verbatim in every cell. "undisclosed" = the provider's documentation does not state a default; "unsupported" = the parameter is not exposed by the endpoint. *For Kimi-K2.5, 1.0 is the documented default for the Thinking mode that Azure serves; the generic request-schema default across modes is 0.6.

G.3. Code Release and Audit Scripts

Every numeric claim in the paper is backed by a self-contained audit script under `tools/` in the released code repository. Each script imports the canonical task-level loader from `tools/_data.py` (see §G.4) so all aggregations operate on the same filtered, normalized data. Running a script prints `paper_value` vs. `computed_value` side-by-side along with an OK / DRIFT / CONTRADICTED verdict, so any drift between the manuscript and the released data can be detected by re-executing the relevant audit. Table 28 maps each script to the section, claim, or table it reproduces.

Script	Reproduces	Invocation
`audit_failure_steps.py`	§4.9, App. F	`python3 tools/audit_failure_steps.py`
`audit_variance.py`	§4.3, Tab. 22	`python3 tools/audit_variance.py`
`audit_within_model_spread.py`	§4.4, Tab. 16	`python3 tools/audit_within_model_spread.py`
`audit_pairwise_significance.py`	App. F: pairwise tables	`python3 tools/audit_pairwise_significance.py`
`audit_spearman.py`	§4.7, Tab. 23	`python3 tools/audit_spearman.py`
`audit_shortlisting.py`	§4.4: shortlisting deltas	`python3 tools/audit_shortlisting.py`
`audit_cost.py`	§4.9: cost-efficiency claims	`python3 tools/audit_cost.py`
`audit_means.py`	§4.1: per-config weighted means	`python3 tools/audit_means.py`
`audit_best_per_benchmark.py`	§4.1: per-benchmark winners	`python3 tools/audit_best_per_benchmark.py`
`audit_stability_std.py`	§4.1: cross-config STD claim	`python3 tools/audit_stability_std.py`
`generate_appendix_tables.py`	batch-regen all `tables/auto_*.tex`	`python3 tools/generate_appendix_tables.py`
`run_exgentic_cli.py`	end-to-end re-run via the Exgentic `analyse` CLI	`python3 tools/run_exgentic_cli.py`

Table 28. Audit scripts released with the paper. Cross-references in the "Reproduces" column point to the corresponding section, claim cluster, or table in this manuscript; some references resolve to the closest available label if no granular label exists. `generate_appendix_tables.py` regenerates every `tables/auto_*.tex` file (`auto_full_leaderboard`, `auto_within_model`, `auto_pairwise_<m>`, `auto_variance`, `auto_spearman`, `auto_data_quality`) in a single call.

G.4. Data and Environment

Data layout. All audits read from a single per-session file `${EXGENTIC_EXPERIMENTS_DIR}/experiments/results/sessions.csv` and a per-run aggregate file `runs.csv` in the same directory. The released paper artifact bundles both files. The canonical loader is:

```
from tools._data import load_tasks
tasks = load_tasks(include_open_models=True)
```

`load_tasks` drops empty rows, the deprecated `appworld_test_normal_old` subset, the `gemini_cli` scaffold (no completed runs), and any directories with `_old` or `_copy` suffixes; remaining rows are normalized to the schema {`model`, `agent`, `bench`, `task_id`, `score`, `success`, `cost`, `steps`}.

Score field convention. Two fields are exposed per task: `score` (continuous in $[0, 1]$, partial credit) and `success` (binary). The Exgentic CLI's `analyse` subcommand explicitly aggregates `benchmark_score` (per-config binary success rate), so paper claims should be checked against the binary `success` field; the variance decomposition, pairwise significance tests, and within-model spread all use `success`.

SWE-Bench is the exception. For SWE-Bench specifically, the session-completion field (`success`) sits near 99%, while the patch-pass criterion ($\texttt{score} \geq 1$) sits near 37%; using `success` would misclassify real failures as successes. The failure-step gap analysis (§4.9, `audit_failure_steps.py`) therefore uses $\texttt{score} \geq 1$ as the success criterion on SWE-Bench and `success` elsewhere. This is the single most important methodology note for any re-run of the failure-step gap.

Bench-weighted average. The headline weighted mean is $\frac{1}{4}$ each for AppWorld, BrowseComp+, and SWE-Bench Verified, and $\frac{1}{12}$ each for the three τ^2-Bench subdomains, so the τ^2 family contributes $\frac{1}{4}$ in total (matching paper §4.1). `tools/_data.py` exports `BENCH_WEIGHTS` and `weighted_mean_over_benches` helpers so every script applies this convention identically.

Closed vs. open-weight subsets. The paper's primary configurations use the closed-only subset (Claude Opus 4.5, Gemini 3, GPT 5.2; $3 \times 5 \times 6 = 90$ cells). `include_open_models=True` opts in to DeepSeek-V3.2 and Kimi-K2.5 for the open-weight extension.

Compute footprint. Total inference spend across the 150 (agent, model, benchmark) cells, paid against provider APIs (AWS Bedrock, Azure OpenAI, GCP Vertex AI, Azure AI Foundry), came to ∼\$20K. Per-task cost varies by roughly 30× across the 15 closed configurations (§4.9, App. E.1, Tab. 10): the efficiency frontier sits at ∼\$2.43 score/\$ (ReAct+GPT 5.2, OpenAI Solo+DeepSeek-V3.2) and the lowest-efficiency cells at ∼\$0.08 score/\$ (Claude Code+Claude Opus 4.5). `audit_cost.py` prints the full per-cell score-and-cost breakdown that backs Tab. 10, so a practitioner can read off the expected API spend for re-running any single cell.

G.5. Task IDs Used per Benchmark

We report below the exact `task_id` values evaluated in every agent–model cell (550 in total: 100 per benchmark; 50 for τ^2-Bench-Airline). These IDs match the `task_id` column of the released per-session results file and uniquely pin which subset of each benchmark was run, independent of any sampling seed or upstream benchmark revision.

SWE-Bench Verified (n=100).

```
astropy__astropy-13453
astropy__astropy-13977
astropy__astropy-14096
astropy__astropy-14369
astropy__astropy-8872
django__django-10973
django__django-11163
django__django-11239
django__django-11299
django__django-11477
django__django-11551
django__django-11749
django__django-11999
```

```
django__django-12143
django__django-12193
django__django-12273
django__django-12754
django__django-12774
django__django-13012
django__django-13023
django__django-13028
django__django-13033
django__django-13089
django__django-13128
django__django-13568
django__django-13794
django__django-13809
django__django-13820
django__django-14017
django__django-14053
django__django-14351
django__django-14404
django__django-14434
django__django-14493
django__django-14559
django__django-14608
django__django-14631
django__django-14672
django__django-14792
django__django-15037
django__django-15103
django__django-15128
django__django-15252
django__django-15277
django__django-15315
django__django-15368
django__django-15569
django__django-15851
django__django-15916
django__django-15930
django__django-15957
django__django-15987
django__django-16082
django__django-16116
django__django-16263
django__django-16454
django__django-16485
django__django-16901
django__django-16938
django__django-17084
django__django-7530
matplotlib__matplotlib-22871
matplotlib__matplotlib-25332
pallets__flask-5014
psf__requests-1766
psf__requests-6028
pydata__xarray-3993
pydata__xarray-4075
pydata__xarray-6938
pylint-dev__pylint-4551
pylint-dev__pylint-6903
pytest-dev__pytest-5262
pytest-dev__pytest-5631
pytest-dev__pytest-7205
pytest-dev__pytest-7521
scikit-learn__scikit-learn-13135
scikit-learn__scikit-learn-14629
scikit-learn__scikit-learn-14710
scikit-learn__scikit-learn-14894
scikit-learn__scikit-learn-25232
sphinx-doc__sphinx-10449
sphinx-doc__sphinx-7440
sphinx-doc__sphinx-7757
sphinx-doc__sphinx-7910
sphinx-doc__sphinx-7985
sphinx-doc__sphinx-8459
sphinx-doc__sphinx-8593
sphinx-doc__sphinx-8595
sphinx-doc__sphinx-9281
sphinx-doc__sphinx-9591
sympy__sympy-11618
sympy__sympy-12096
sympy__sympy-13551
sympy__sympy-13852
```

```
sympy__sympy-13974
sympy__sympy-14248
sympy__sympy-15976
sympy__sympy-17655
sympy__sympy-19040
sympy__sympy-20916
```

AppWorld (n=100)**.**

```
024c982_1
024c982_3
042a9fc_2
09b0ee6_2
09b0ee6_3
0a9d82a_1
0d01c76_3
0de03ea_2
1150ed6_3
166f4ff_1
166f4ff_2
21abae1_2
21abae1_3
270f1ff_1
270f1ff_3
29a7b7e_1
29a7b7e_2
29a7b7e_3
2d9f728_1
2d9f728_3
31dc501_1
31dc501_2
31dc501_3
325d6ec_2
32616b5_2
3aa1a22_2
3aa1a22_3
3b8fb7a_1
3b8fb7a_2
3d9a636_2
3d9a636_3
425a494_1
425a494_3
522e5e5_1
552869a_1
552869a_2
59fae45_1
59fae45_2
5a83b05_1
5a83b05_2
634f342_1
634f342_3
652485c_2
6b6ca61_1
6b6ca61_2
6b6ca61_3
6f4b9a5_3
7847649_1
7847649_3
83a7951_2
83a7951_3
8749218_2
8749218_3
8ce6779_1
8ce6779_3
9016950_2
9016950_3
90adc3f_1
90adc3f_2
90adc3f_3
986aa4e_2
986aa4e_3
9dabbc9_2
9dabbc9_3
9ef798c_2
9ef798c_3
a30375d_1
a30375d_2
a30375d_3
afc4005_1
afc4005_3
```

```
b6d1104_2
b9c5c9a_1
b9c5c9a_2
bde252e_2
bde252e_3
c77c005_2
c77c005_3
ccf4b82_1
ccf4b82_3
cef9191_1
cef9191_2
d18139b_1
d18139b_2
d18139b_3
d194965_2
d194965_3
d6ac34d_1
d6ac34d_2
d6ac34d_3
dac78d9_1
dac78d9_2
dac78d9_3
f323bae_3
f3f60f0_2
f3f60f0_3
f861c32_1
fd1f8fa_1
fd1f8fa_3
ff58e36_1
```

BrowseComp+ (n=100).

```
6, 8, 23, 85, 97, 127, 134, 152, 169, 177,
184, 200, 201, 209, 237, 242, 251, 270, 315, 327,
337, 347, 362, 372, 395, 413, 418, 438, 443, 469,
491, 498, 519, 524, 553, 555, 558, 562, 571, 581,
582, 587, 591, 592, 598, 601, 618, 620, 637, 650,
661, 666, 672, 675, 686, 707, 714, 717, 722, 730,
756, 771, 772, 785, 801, 810, 828, 830, 834, 873,
876, 882, 898, 930, 947, 960, 969, 984, 991, 992,
1021, 1058, 1079, 1096, 1121, 1147, 1161, 1162, 1176, 1191,
1198, 1203, 1208, 1211, 1212, 1220, 1227, 1237, 1239, 1264
```

τ^2-Bench-Airline (n=50). IDs `0` through `49` (the full benchmark task pool, in order).

τ^2-Bench-Retail (n=100).

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 16, 17, 18, 19, 22, 23,
24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
34, 35, 36, 37, 38, 39, 40, 41, 42, 43,
46, 47, 48, 49, 50, 51, 52, 53, 55, 56,
57, 58, 59, 60, 61, 63, 64, 65, 66, 68,
69, 70, 71, 72, 75, 77, 78, 79, 80, 81,
82, 83, 84, 85, 86, 87, 88, 89, 90, 91,
92, 93, 94, 95, 96, 97, 98, 99, 100, 101,
102, 103, 104, 106, 107, 108, 109, 110, 112, 113
```

τ^2-Bench-Telecom (n=100).

```
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_apn_mms_setting|break_app_both_permissions|data_mode_off|
data_usage_exceeded|unseat_sim_card|user_abroad_roaming_disabled_off[PERSONA:Hard]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_apn_mms_setting|break_app_both_permissions|data_mode_off|
data_usage_exceeded|unseat_sim_card|user_abroad_roaming_disabled_on[PERSONA:Hard]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_apn_mms_setting|break_app_both_permissions|unseat_sim_card|
user_abroad_roaming_enabled_off[PERSONA:Hard]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_apn_mms_setting|break_app_sms_permission|data_mode_off|
data_usage_exceeded|unseat_sim_card|user_abroad_roaming_disabled_on[PERSONA:None]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_apn_mms_setting|break_app_sms_permission|data_mode_off|
data_usage_exceeded|unseat_sim_card|user_abroad_roaming_enabled_off[PERSONA:None]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_apn_mms_setting|break_app_storage_permission|data_mode_off|
data_usage_exceeded|unseat_sim_card[PERSONA:Easy]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_apn_mms_setting|break_app_storage_permission|data_mode_off|
data_usage_exceeded|unseat_sim_card|user_abroad_roaming_disabled_off[PERSONA:Easy]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_app_sms_permission|data_mode_off|data_usage_exceeded|
unseat_sim_card|user_abroad_roaming_disabled_off[PERSONA:None]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_app_storage_permission|data_mode_off|data_usage_exceeded|
unseat_sim_card|user_abroad_roaming_enabled_off[PERSONA:Hard]
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|break_app_storage_permission|unseat_sim_card[PERSONA:Hard]
```

```
[mms_issue]airplane_mode_on|bad_network_preference|bad_wifi_calling|data_usage_exceeded|unseat_sim_card|
user_abroad_roaming_disabled_on[PERSONA:Easy]
[mms_issue]airplane_mode_on|bad_network_preference|break_apn_mms_setting|break_app_both_permissions|data_mode_off|data_usage_exceeded
|user_abroad_roaming_enabled_off[PERSONA:None]
[mms_issue]airplane_mode_on|bad_network_preference|break_apn_mms_setting|break_app_storage_permission[PERSONA:None]
[mms_issue]airplane_mode_on|bad_network_preference|break_apn_mms_setting|data_mode_off|unseat_sim_card|
user_abroad_roaming_disabled_off[PERSONA:Easy]
[mms_issue]airplane_mode_on|bad_network_preference|break_apn_mms_setting|data_usage_exceeded[PERSONA:Hard]
[mms_issue]airplane_mode_on|bad_network_preference|break_app_both_permissions|data_usage_exceeded|unseat_sim_card|
user_abroad_roaming_disabled_on[PERSONA:Hard]
[mms_issue]airplane_mode_on|bad_network_preference|break_app_storage_permission|data_mode_off|user_abroad_roaming_enabled_off[PERSONA
:Easy]
[mms_issue]airplane_mode_on|bad_wifi_calling|break_app_both_permissions|data_mode_off|data_usage_exceeded|unseat_sim_card|
user_abroad_roaming_enabled_off[PERSONA:None]
[mms_issue]airplane_mode_on|bad_wifi_calling|user_abroad_roaming_enabled_off[PERSONA:Easy]
[mms_issue]airplane_mode_on|break_app_both_permissions[PERSONA:Hard]
[mms_issue]airplane_mode_on|break_app_both_permissions|data_usage_exceeded|user_abroad_roaming_disabled_off[PERSONA:None]
[mms_issue]bad_network_preference|bad_wifi_calling|break_apn_mms_setting|break_app_storage_permission|data_mode_off|
data_usage_exceeded|unseat_sim_card[PERSONA:Easy]
[mms_issue]bad_network_preference|bad_wifi_calling|break_app_both_permissions|data_usage_exceeded|unseat_sim_card|
user_abroad_roaming_disabled_off[PERSONA:None]
[mms_issue]bad_network_preference|bad_wifi_calling|break_app_both_permissions|data_usage_exceeded|user_abroad_roaming_disabled_off[
PERSONA:Hard]
[mms_issue]bad_network_preference|bad_wifi_calling|break_app_sms_permission|data_mode_off|data_usage_exceeded|unseat_sim_card|
user_abroad_roaming_enabled_off[PERSONA:Easy]
[mms_issue]bad_network_preference|bad_wifi_calling|break_app_sms_permission|data_mode_off|unseat_sim_card|
user_abroad_roaming_enabled_off[PERSONA:Hard]
[mms_issue]bad_network_preference|break_app_both_permissions[PERSONA:Easy]
[mms_issue]bad_network_preference|break_app_sms_permission|data_mode_off|data_usage_exceeded|user_abroad_roaming_enabled_off[PERSONA:
None]
[mms_issue]bad_network_preference|break_app_sms_permission|user_abroad_roaming_disabled_on[PERSONA:Hard]
[mms_issue]bad_network_preference|user_abroad_roaming_disabled_off[PERSONA:None]
[mms_issue]bad_wifi_calling|break_apn_mms_setting|break_app_both_permissions|data_mode_off|data_usage_exceeded|
user_abroad_roaming_disabled_off[PERSONA:None]
[mms_issue]bad_wifi_calling|break_apn_mms_setting|break_app_sms_permission|data_usage_exceeded[PERSONA:Hard]
[mms_issue]bad_wifi_calling|break_apn_mms_setting|data_mode_off|data_usage_exceeded|unseat_sim_card[PERSONA:None]
[mms_issue]bad_wifi_calling|break_apn_mms_setting|unseat_sim_card|user_abroad_roaming_enabled_off[PERSONA:Easy]
[mms_issue]break_apn_mms_setting|data_mode_off|data_usage_exceeded|user_abroad_roaming_disabled_on[PERSONA:Hard]
[mms_issue]break_apn_mms_setting|data_mode_off|user_abroad_roaming_disabled_on[PERSONA:Hard]
[mms_issue]break_apn_mms_setting|user_abroad_roaming_enabled_off[PERSONA:Hard]
[mms_issue]break_app_both_permissions|data_usage_exceeded[PERSONA:Hard]
[mms_issue]break_app_both_permissions|unseat_sim_card|user_abroad_roaming_disabled_on[PERSONA:Hard]
[mms_issue]break_app_sms_permission|data_mode_off[PERSONA:None]
[mms_issue]break_app_sms_permission|data_usage_exceeded|user_abroad_roaming_disabled_on[PERSONA:None]
[mms_issue]break_app_storage_permission|data_usage_exceeded[PERSONA:Easy]
[mms_issue]break_app_storage_permission|unseat_sim_card|user_abroad_roaming_disabled_on[PERSONA:Easy]
[mobile_data_issue]airplane_mode_on|bad_network_preference[PERSONA:Hard]
[mobile_data_issue]airplane_mode_on|bad_network_preference|bad_vpn|data_mode_off|data_saver_mode_on|data_usage_exceeded|
user_abroad_roaming_disabled_off[PERSONA:Hard]
[mobile_data_issue]airplane_mode_on|bad_network_preference|bad_vpn|data_mode_off|data_saver_mode_on|data_usage_exceeded|
user_abroad_roaming_disabled_on[PERSONA:None]
[mobile_data_issue]airplane_mode_on|bad_network_preference|bad_vpn|data_mode_off|data_saver_mode_on|data_usage_exceeded|
user_abroad_roaming_enabled_off[PERSONA:Easy]
[mobile_data_issue]airplane_mode_on|bad_network_preference|bad_vpn|data_mode_off|data_saver_mode_on|user_abroad_roaming_disabled_off[
PERSONA:Hard]
[mobile_data_issue]airplane_mode_on|bad_network_preference|bad_vpn|data_mode_off|data_usage_exceeded|user_abroad_roaming_enabled_off[
PERSONA:Easy]
[mobile_data_issue]airplane_mode_on|bad_network_preference|bad_vpn|data_saver_mode_on|data_usage_exceeded|
user_abroad_roaming_enabled_off[PERSONA:None]
[mobile_data_issue]airplane_mode_on|bad_network_preference|bad_vpn|data_saver_mode_on|user_abroad_roaming_disabled_on[PERSONA:Easy]
[mobile_data_issue]airplane_mode_on|bad_network_preference|data_mode_off|data_saver_mode_on[PERSONA:Hard]
[mobile_data_issue]airplane_mode_on|bad_network_preference|data_mode_off|data_saver_mode_on|data_usage_exceeded|
user_abroad_roaming_disabled_on[PERSONA:None]
[mobile_data_issue]airplane_mode_on|bad_network_preference|data_mode_off|data_usage_exceeded|user_abroad_roaming_disabled_on[PERSONA:
None]
[mobile_data_issue]airplane_mode_on|bad_network_preference|data_mode_off|user_abroad_roaming_disabled_on[PERSONA:Hard]
[mobile_data_issue]airplane_mode_on|bad_network_preference|data_saver_mode_on|data_usage_exceeded|user_abroad_roaming_disabled_off[
PERSONA:None]
[mobile_data_issue]airplane_mode_on|data_mode_off[PERSONA:None]
[mobile_data_issue]airplane_mode_on|data_mode_off|data_saver_mode_on|data_usage_exceeded|user_abroad_roaming_enabled_off[PERSONA:Hard
]
[mobile_data_issue]airplane_mode_on|data_saver_mode_on|user_abroad_roaming_disabled_on[PERSONA:None]
[mobile_data_issue]airplane_mode_on|user_abroad_roaming_enabled_off[PERSONA:None]
[mobile_data_issue]bad_network_preference|bad_vpn|data_mode_off|data_saver_mode_on[PERSONA:None]
[mobile_data_issue]bad_network_preference|bad_vpn|data_mode_off|data_saver_mode_on|data_usage_exceeded|
user_abroad_roaming_enabled_off[PERSONA:Easy]
[mobile_data_issue]bad_network_preference|bad_vpn|data_saver_mode_on|data_usage_exceeded|user_abroad_roaming_disabled_off[PERSONA:
Easy]
[mobile_data_issue]bad_network_preference|bad_vpn|user_abroad_roaming_disabled_off[PERSONA:Hard]
[mobile_data_issue]bad_network_preference|bad_vpn|user_abroad_roaming_disabled_on[PERSONA:None]
[mobile_data_issue]bad_network_preference|bad_vpn|user_abroad_roaming_enabled_off[PERSONA:Easy]
[mobile_data_issue]bad_network_preference|data_mode_off|data_saver_mode_on|data_usage_exceeded|user_abroad_roaming_disabled_off[
PERSONA:Easy]
[mobile_data_issue]bad_network_preference|data_saver_mode_on|data_usage_exceeded[PERSONA:Hard]
[mobile_data_issue]bad_network_preference|user_abroad_roaming_enabled_off[PERSONA:Hard]
[mobile_data_issue]bad_vpn|data_mode_off|data_usage_exceeded|user_abroad_roaming_disabled_off[PERSONA:None]
[mobile_data_issue]bad_vpn|data_saver_mode_on|user_abroad_roaming_disabled_on[PERSONA:None]
[mobile_data_issue]data_mode_off|data_usage_exceeded|user_abroad_roaming_disabled_off[PERSONA:Hard]
[mobile_data_issue]data_saver_mode_on|data_usage_exceeded[PERSONA:Easy]
[mobile_data_issue]data_saver_mode_on|user_abroad_roaming_enabled_off[PERSONA:Easy]
[mobile_data_issue]data_usage_exceeded|user_abroad_roaming_enabled_off[PERSONA:Easy]
[service_issue]airplane_mode_on|break_apn_settings|contract_end_suspension|lock_sim_card_pin[PERSONA:None]
```

```
[service_issue]airplane_mode_on|break_apn_settings|contract_end_suspension|lock_sim_card_pin|unseat_sim_card[PERSONA:Easy]
[service_issue]airplane_mode_on|break_apn_settings|contract_end_suspension|unseat_sim_card[PERSONA:Easy]
[service_issue]airplane_mode_on|break_apn_settings|lock_sim_card_pin[PERSONA:None]
[service_issue]airplane_mode_on|break_apn_settings|lock_sim_card_pin|overdue_bill_suspension[PERSONA:Hard]
[service_issue]airplane_mode_on|break_apn_settings|lock_sim_card_pin|overdue_bill_suspension|unseat_sim_card[PERSONA:None]
[service_issue]airplane_mode_on|break_apn_settings|lock_sim_card_pin|unseat_sim_card[PERSONA:None]
[service_issue]airplane_mode_on|break_apn_settings|overdue_bill_suspension[PERSONA:None]
[service_issue]airplane_mode_on|break_apn_settings|unseat_sim_card[PERSONA:None]
[service_issue]airplane_mode_on|contract_end_suspension|lock_sim_card_pin|unseat_sim_card[PERSONA:Hard]
[service_issue]airplane_mode_on|lock_sim_card_pin[PERSONA:Easy]
[service_issue]airplane_mode_on|lock_sim_card_pin|overdue_bill_suspension[PERSONA:Easy]
[service_issue]airplane_mode_on|lock_sim_card_pin|overdue_bill_suspension|unseat_sim_card[PERSONA:Easy]
[service_issue]airplane_mode_on|lock_sim_card_pin|unseat_sim_card[PERSONA:Hard]
[service_issue]airplane_mode_on|overdue_bill_suspension[PERSONA:None]
[service_issue]airplane_mode_on|overdue_bill_suspension|unseat_sim_card[PERSONA:Easy]
[service_issue]airplane_mode_on|unseat_sim_card[PERSONA:None]
[service_issue]break_apn_settings|lock_sim_card_pin[PERSONA:None]
[service_issue]break_apn_settings|lock_sim_card_pin|overdue_bill_suspension[PERSONA:Easy]
[service_issue]break_apn_settings|lock_sim_card_pin|overdue_bill_suspension|unseat_sim_card[PERSONA:Easy]
[service_issue]break_apn_settings|overdue_bill_suspension|unseat_sim_card[PERSONA:Hard]
[service_issue]contract_end_suspension|lock_sim_card_pin[PERSONA:Hard]
[service_issue]contract_end_suspension|unseat_sim_card[PERSONA:Hard]
[service_issue]lock_sim_card_pin|overdue_bill_suspension[PERSONA:Easy]
[service_issue]overdue_bill_suspension|unseat_sim_card[PERSONA:Easy]
```

H. Behavioral Error Analysis on Agentic Trajectories

This appendix characterizes how general agents fail. We adapt ERRORMAP (Ashury-Tahan et al., 2026) to long-horizon trajectories and produce 27 behavioral failure categories over all failed sessions with full trajectory data, validated on a 100-record sample. The analysis is descriptive: it documents the shape of the failure surface in our corpus and is not powered for causal claims about model, architecture, or benchmark effects.

H.1. Setup

We adapt ERRORMAP to agent trajectories produced by the Exgentic, using the five agent architectures (Claude Code, ReAct, ReAct Short, Smolagent, OpenAI Solo), five LM backbones (Claude Opus 4.5, DeepSeek-V3.2, Gemini 3 Pro, GPT 5.2, Kimi-K2.5), and six benchmarks (AppWorld, BrowseComp+, SWE-Bench Verified, and three τ^2-Bench variants) as the leaderboard. We retain all failed sessions with trajectory data. Zero-step sessions (run-level orchestrator failures, including the cases that dominate the open-weight architecture sink on τ^2-Bench in §4.7) lack trajectories and are excluded by construction. Of the retained sessions, 76.7% have a peer-success "gold" trajectory from another architecture or model on the same task, used as the judge's reference.

The port required only a trajectory-to-CSV adapter and a QA→trajectory vocabulary refresh of the prompt templates; ERRORMAP's clustering pipeline, schema, and output format are unchanged. We use `gpt-5.5` as the judge with the upstream default `max_num_clusters=25`, and run end-to-end in 42 minutes for $246. The first run produced 26 top-level categories with *Other* = 25.8%; we manually linked synonyms (released as a deterministic mapping) to reach the final 27 categories with *Other* = 2.1% (ERRORMAP reports 4.8%). The adapted code and all analysis artifacts are released.

H.2. Distribution of Failure Categories

The largest category, *Premature Termination*, accounts for 8.1% of categorized records (n = 2,868); the top-10 cumulatively cover 57.5% (Table 29, Figure 6). Of the 27 top-level categories, 22 cross 1% prevalence (Table 29); the remaining five plus the *Other* bucket cover the long tail. Several categories are heavily concentrated in a single benchmark (e.g., *Diagnostic Protocol Error* is 97% τ^2-Bench-Telecom; *Search Recovery & Adaptation* is 92% BrowseComp+), so the prevalence figures reflect benchmark mix in our corpus.

H.3. Per-Model Profiles

Failure-category shares vary across the five backbone models (Figure 7). Reporting categories where a model's share differs from the 5-model grand mean by at least 1.5 percentage points: DeepSeek-V3.2 over-represents *Search Recovery & Adaptation* (+4.0pp), *Premature Termination* (+3.4pp), and *Constraint Interpretation Error* (+2.3pp). Kimi-K2.5 over-represents *Tool Invocation Error* (+3.6pp), *Unsupported Fabrication* (+3.1pp), and *Action Execution Error* (+2.4pp). GPT 5.2 shows the largest single deviation in this analysis on *Escalation Handling* (+4.6pp), followed by *Premature Termination* (+3.8pp); it also has the lowest *Unsupported Fabrication* share of any model (−4.3pp from mean). Claude Opus

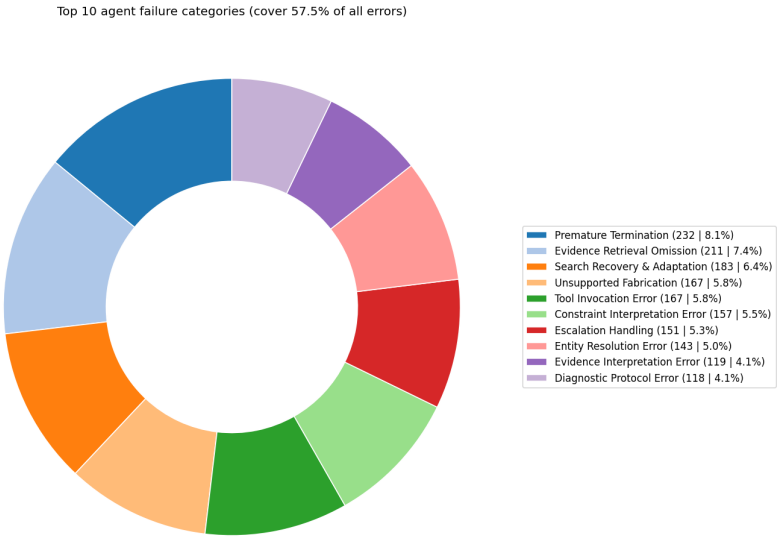

Figure 6. Top-10 categories by share of categorized records ($n = 2{,}868$).

4.5 over-represents *Candidate Verification Failure* (+3.0pp), *Remediation Omission* (+2.7pp), and *Diagnostic Protocol Error* (+2.0pp). Gemini 3 Pro over-represents *Unsupported Fabrication* (+2.1pp), *Evidence Retrieval Omission* (+2.0pp), and *Confirmation Handling Error* (+1.8pp).

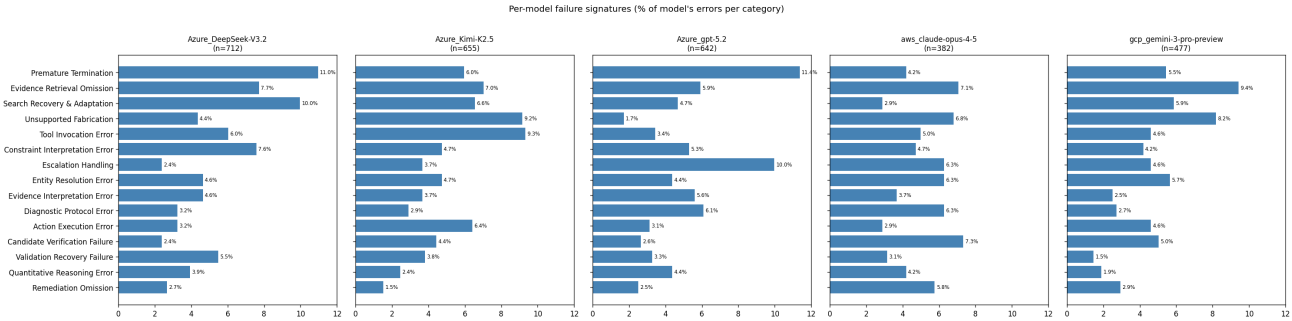

Figure 7. Per-model failure-category shares (top-15 categories, column-normalized).

H.4. Per-Architecture Profiles

Architecture-level shares vary along a different axis than the model axis (Figure 8). The autonomous architectures Claude Code and OpenAI Solo over-represent *Premature Termination* relative to the 5-architecture mean of 8.7% (+4.0pp and +7.3pp respectively); OpenAI Solo also over-represents *Finalization Protocol Error* (+1.8pp). The tool-calling architectures ReAct and ReAct Short sit at the opposite end on *Premature Termination* (−5.0pp and −3.7pp); both over-represent *Evidence Retrieval Omission* (+2.1pp and +1.5pp). Smolagent, the code-execution architecture, has a distinct profile with elevated *Search Recovery & Adaptation* (+3.5pp), *Authentication Handling Error* (+1.9pp), and *Evidence Interpretation Error* (+1.7pp). On this dependent variable, the architecture axis produces non-trivial within-axis spread, in contrast to its small contribution to success-rate variance ($\eta^2 = 0.5\%$, §4.3).

H.5. Per-Benchmark Profiles

Benchmark-level shares show the largest within-axis spread of the three axes (Figure 9). BrowseComp+ over-represents *Search Recovery & Adaptation* (+17.3pp, the largest single deviation in this analysis), *Candidate Verification Failure* (+8.4pp), and *Evidence Retrieval Omission* (+6.1pp). SWE-Bench Verified over-represents *Action Execution Er-*

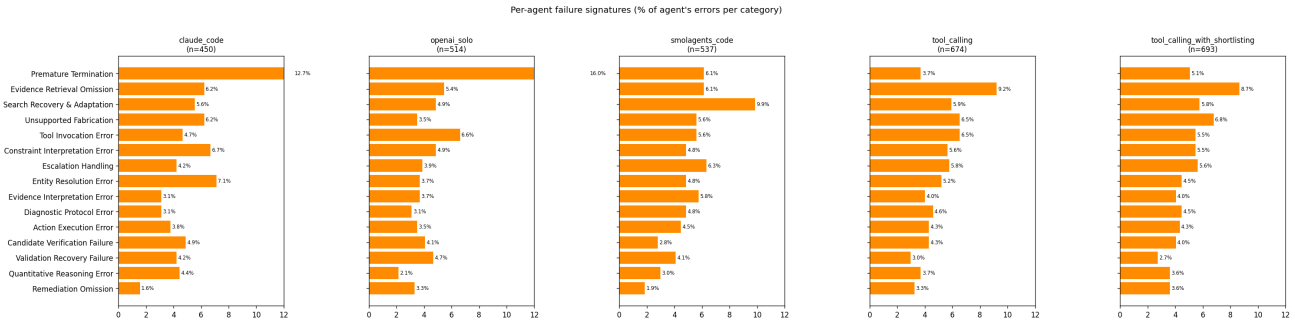

Figure 8. Per-architecture failure-category shares (top-15 categories, column-normalized).

ror (+14.9pp) and *Premature Termination* (+14.3pp). τ^2-Bench-Telecom over-represents *Diagnostic Protocol Error* (+19.0pp), *Escalation Handling* (+15.9pp), and *Remediation Omission* (+8.8pp). τ^2-Bench-Airline over-represents *Policy Eligibility Error* (+11.9pp), *Quantitative Reasoning Error* (+8.5pp), and *Constraint Interpretation Error* (+5.0pp). τ^2-Bench-Retail over-represents *Confirmation Handling Error* (+6.6pp), *Unsupported Fabrication* (+4.7pp), and *Constraint Interpretation Error* (+4.1pp). AppWorld over-represents *Tool Invocation Error* (+11.1pp), *Validation Recovery Failure* (+9.3pp), and *Coverage Enumeration Omission* (+6.2pp), and shows the broadest spread (seven categories at $\geq$ +2pp). Per-benchmark sample sizes are unequal: $n = 125$ for SWE-Bench Verified, $n = 341$ for τ^2-Bench-Airline, $n = 441$ for τ^2-Bench-Retail, $n = 486$ for τ^2-Bench-Telecom, $n = 687$ for AppWorld, $n = 788$ for BrowseComp+; SWE-Bench Verified carries the largest uncertainty.

Figure 9. Per-benchmark failure-category shares (top-15 categories, column-normalized).

H.6. Validation

Following ERRORMAP's protocol, we sampled 100 labelled records and presented the judge with the assigned category vs. a random alternative; the sample over-represents *Other* (27/100) to stress-test the residual bucket. The judge prefers the assigned label in 85/100 cases overall. Broken down, agreement is 66/73 (90%) for named categories and 19/27 (70%) for *Other*; misclassification concentrates in the residual bucket.

H.7. Category Definitions

Table 30 lists the natural-language definition of each top-level failure category, as produced by the LLM judge during the recursive-categorization stage and used as the classification reference.

Table 30. Definitions of the 27 top-level failure categories produced by the recursive categorization stage. Definitions are the LLM judge's own outputs (saved at category-construction time); the analysis records released alongside the paper include the raw responses.

Category	Definition
Action Execution Error	Agent has authorization and correct intent but carries out required mutations, payments, bookings, patches, cleanup, batching, or edits incorrectly or incompletely.

Continued on next page

(Table 30 continued)

Category	Definition
Authentication Handling Error	Agent omits, misreads, invents, corrupts, extracts incorrectly, or misuses credentials, tokens, login state, authentication order, or authenticated context.
Candidate Verification Failure	Agent commits to or rejects a candidate without checking discriminating constraints, alternatives, contradictions, disqualifiers, or uncertainty against available evidence.
Confirmation Handling Error	Agent omits, mistimes, misreads, repeats, or malforms required consent, clarification, preference check, disclosure, approval, or post-action confirmation communication.
Constraint Interpretation Error	Agent misreads explicit user or task constraints, including scope, dates, filters, preferences, privacy requirements, answer form, variants, or replacement boundaries.
Coverage Enumeration Omission	Agent fails to exhaust required pages, lists, histories, variants, batches, statuses, transactions, recommendations, options, contacts, or libraries for complete coverage.
Diagnostic Protocol Error	Agent skips, misorders, detours from, or misinterprets required troubleshooting checks, reproductions, account tests, connectivity tests, configuration checks, or diagnostic verification.
Entity Resolution Error	Agent identifies or acts on the wrong entity, such as a person, account, order, item, address, path, role, or counterparty.
Escalation Handling	Agent fails to escalate when required, or performs required escalation with missing tools, handoff details, notifications, routing, or protocol steps. Combined with: Agent escalates or transfers too early, before available in-scope troubleshooting, diagnostics, modification, cancellation, policy decision, or self-service resolution is attempted.
Evidence Interpretation Error	Agent misunderstands retrieved evidence, including source details, filters, tool outputs, rankings, attributions, or implications, after the evidence is available.
Evidence Retrieval Omission	Agent fails to open, inspect, expand, download, extract, or verify a necessary source, file, message, prior state, or record.
Finalization Protocol Error	Agent completes work but omits or malforms required final answer, submission, completion signal, exact-answer field, alias handling, or answer format.
Hard to Analyze	Failure label is ambiguous, subjective, null, artifact-like, or lacks enough context to infer a specific failed skill confidently.
Instruction Following Error	Agent ignores, misapplies, or contradicts explicit user, system, or task instructions.
Looping Behavior	Agent repeats actions, reasoning, or tool calls without making progress toward completion.
Planning Error	Agent lacks a viable step sequence, decomposes poorly, or chooses a strategy that prevents completion.
Policy Eligibility Error	Agent misapplies policy rules governing eligibility, approval, refusal, compensation, cancellation, exchange, refund, return, insurance, payments, or in-scope service.
Premature Termination	Agent stops or responds as finished while required investigation, recovery, processing, implementation, troubleshooting, cleanup, or resolution work remains incomplete.
Quantitative Reasoning Error	Agent incorrectly extracts, counts, ranks, compares, allocates, or calculates numeric values such as prices, refunds, fares, balances, durations, quantities, or availability.
Query Formulation Error	Agent constructs missing, malformed, overbroad, overconstrained, literal, or poorly targeted queries, causing relevant information to be excluded from retrieval.
Remediation Omission	Agent diagnoses or identifies a required fix but fails to perform necessary remediation, such as reset, update, enablement, permission, configuration, or cleanup.
Search Recovery & Adaptation	Agent does not handle retrieval failures, unavailable sources, invalid queries, timeouts, or empty results through retries, repair, or alternate sources. Combined with: Agent persists with unproductive search strategy instead of pivoting terms, decomposing clues, changing scope, switching sources, or following promising leads.

Continued on next page

(Table 30 continued)

Category	Definition
State/Goal Tracking Error	Agent loses the user's objective, switches tasks, or optimizes for the wrong success condition. / Agent loses track of completed actions, environment state, or unresolved subtasks during execution. / Agent forgets or misuses prior conversation details needed for later decisions.
Tool Invocation Error	Agent chooses an inappropriate tool or violates tool schema, arguments, parameters, capability limits, batching rules, placeholders, call order, or one-tool-per-turn constraints.
Unauthorized Action Execution	Agent performs an action that is unapproved, ineligible, irreversible, unnecessary, overbroad, duplicate, out-of-scope, or not requested by the user.
Unsupported Fabrication	Agent states or relies on facts, source content, credentials, diagnostics, procedures, restrictions, or causal explanations not supported by evidence.
User Communication Error	Agent gives unclear, incomplete, misleading, or poorly scoped communication to the user. Also includes: Agent proceeds despite missing essential user input that should have been requested first.
Validation Recovery Failure	Agent receives tool validation feedback but repeats invalid calls, makes ineffective repairs, or fails to choose a valid call or alternative path.

I. Limitations

While Exgentic provides a clear methodology and reusable building blocks for adaptation, familiarity with these capabilities and additional development work are still required when integrating new agents or benchmarks.

The agent protocols we evaluate (tool-calling, MCP, Python code generation) all accept multimodal inputs, but the benchmarks in this study (widely adopted by frontier-model developers) are selected to stress cross-domain capability rather than multimodal processing. Extending the evaluation to environments with continuous action spaces, such as pixel-level computer-use or robotic control, is a natural next step and will require additions to the Unified Protocol's current typed-action model.

Agent evaluation is expensive, especially for general-purpose agents that must be tested across many benchmarks. Due to cost constraints, our selection of agents and models is limited and does not cover the full range of open-source models or existing general-purpose agents. Three specific scope caveats follow from this. First, the open-weight tier consists of two checkpoints (DeepSeek-V3.2, Kimi-K2.5); claims about "open-weight generality sinks" should be read as observations about these two models, not about open-weight checkpoints in general. Second, each (architecture, model, benchmark) cell is run once with the provider's documented default sampling settings; the resulting per-cell scores reflect a single-sample point estimate of an inherently stochastic process, and per-cell variability across reruns is not measured here. Third, success is judged by each benchmark's native automated evaluator; we do not conduct human evaluation of partial-credit outputs, which is particularly relevant on BrowseComp+ and SWE-Bench Verified where benchmark-native scoring may diverge from human judgment. To enable further progress in the field, future work should therefore explore techniques such as intelligent sampling and early stopping to reduce evaluation costs when it is clear that certain agent–model combinations underperform.

Table 29. Top-level agent failure categories with $\geq$1% prevalence (n = 2,868 categorized records).

#	Category	n	%
1	Premature Termination	232	8.1
2	Evidence Retrieval Omission	211	7.4
3	Search Recovery & Adaptation	183	6.4
4	Unsupported Fabrication	167	5.8
5	Tool Invocation Error	167	5.8
6	Constraint Interpretation Error	157	5.5
7	Escalation Handling	151	5.3
8	Entity Resolution Error	143	5.0
9	Evidence Interpretation Error	119	4.1
10	Diagnostic Protocol Error	118	4.1
11	Action Execution Error	118	4.1
12	Candidate Verification Failure	115	4.0
13	Validation Recovery Failure	104	3.6
14	Quantitative Reasoning Error	97	3.4
15	Remediation Omission	81	2.8
16	Coverage Enumeration Omission	75	2.6
17	Finalization Protocol Error	73	2.5
18	Unauthorized Action Execution	69	2.4
19	Policy Eligibility Error	67	2.3
20	Query Formulation Error	60	2.1
21	Confirmation Handling Error	43	1.5
22	Authentication Handling Error	36	1.3
	Other (uncategorized)	61	2.1