GENERAL AGENT EVALUATION

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

ABSTRACT

General-purpose agents perform tasks in unfamiliar environments without domain-specific customization. Yet no study has systematically measured how agent architecture contributes to their performance across heterogeneous protocols and diverse 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 restrict agents to a single protocol class (web for BrowserGym, CLI for Harbor) or require manual per-benchmark wiring; and benchmarks themselves assume a human integrator who customizes the agent for each benchmark. We close these gaps with three contributions: (1) the Unified Protocol, a benchmark-agent mediation layer derived from existing interaction patterns; (2) Exgentic, an evaluation harness that surfaces benchmarks to any general-purpose agent paired with any backbone model; and (3) the first Open General Agent Leaderboard, a full factorial over five agents $\times$ three LLMs $\times$ six benchmarks spanning software engineering, customer service, deep research, and personal assistance. Our key findings are: (i) general agents adapt across all five domains with no per-domain customization, producing non-trivial performance; (ii) model choice dominates variance 85-fold over agent architecture, yet agent choice still swings results up to 11 percentage points within a single model; (iii) on more than half of the benchmarks, general agents match or beat top published domain-specific scores. We release everything at `www.exgentic.ai`.

1 INTRODUCTION

The field of AI agents has witnessed remarkable progress, with agentic systems demonstrating impressive capabilities across diverse domains, from solving software engineering tasks to navigating web interfaces (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 to the agent the current goals and environment semantics (Fig. 2(B)). 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).

This work starts from three observations about the current state of agent evaluation (Bandel et al., 2026b). *(i) A research gap:* no prior study evaluates the same unmodified agent across multiple

*Correspondence to: `elron.bandel@ibm.com`

#	General Agent	Model	Avg Success	Avg Cost
1	OpenAI Solo	Claude Opus 4.5	0.73	$8.5
2	Claude Code	Claude Opus 4.5	0.67	$8.0
3	Smolagent	Claude Opus 4.5	0.66	$4.4
4	ReAct Short	Gemini 3	0.62	$0.7
5	ReAct Short	Claude Opus 4.5	0.62	$3.8
6	ReAct	Gemini 3	0.61	$0.8
7	ReAct	Claude Opus 4.5	0.61	$5.8
8	OpenAI Solo	Gemini 3	0.60	$2.8
9	Claude Code	Gemini 3	0.57	$2.5
10	Smolagent	Gemini 3	0.56	$1.8
11	ReAct Short	GPT 5.2	0.46	$0.3
12	ReAct	GPT 5.2	0.41	$0.2
13	OpenAI Solo	GPT 5.2	0.39	$0.2
14	Claude Code	GPT 5.2	0.38	$0.4
15	Smolagent	GPT 5.2	0.38	$0.4

Table 1: The first Open General Agent Leaderboard, comparing the same unmodified general agents across standardized benchmarks. Average Success is the mean success rate ($n = 550$ per configuration); Average Cost is the mean cost per task. Per-benchmark scores at $n = 50$–100 have 95% Wilson confidence-interval (CI) half-widths of $\pm$7–9.5pp; model-level comparisons use $n = 2{,}750$ and are correspondingly tighter (App. F). Full per-benchmark scores: Appendix E, Table 5.

domain-specific benchmarks; each benchmark is tested with its own agent, so agent contribution cannot be isolated. *(ii) A framework gap:* no single agent can be dropped onto a new benchmark unchanged, because even the most mature existing harnesses (most notably Inspect (AI Security Institute, UK, 2024), which consolidates sandboxing, logging, and scoring) still require the evaluation author to manually choose a tool-set, sandbox, and solver per benchmark. *(iii) A benchmark gap:* even if such a harness existed, contemporary benchmarks are not ready for it; they encode task information, affordances, and expected interaction assuming a human integrator, not a general-agent interface.

Building on these observations and the emerging case for general agents as a research target (Bandel et al., 2026a), this is the first systematic study to *materialize* it, closing the three gaps above simultaneously with a concrete evaluation method and the first systematic analysis of general agents across diverse environments (Fig. 3). 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 (Fig. 2(C)) 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 $22K (Table 1).

Our analysis of the Open General Agent Leaderboard yields three 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 (Fig. 3), with no per-domain engineering, producing non-trivial performance on every domain. **(2) The model matters most, but so does the agent:** model choice dominates variance 85-fold over agent architecture, yet agent choice still swings results up to 11 percentage points within a single model (Fig. 1). **(3) General agents already rival specialists:** on more than half of tested benchmarks, general agents match or beat top published domain-specific scores (Tab. 2).

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 novel evaluation protocols.

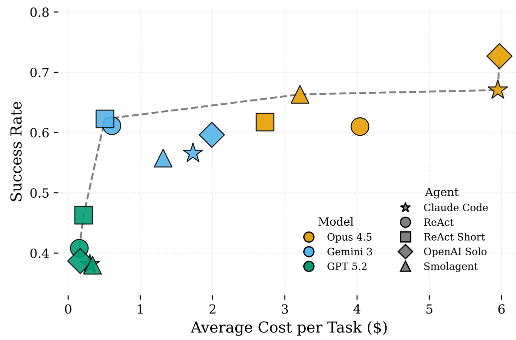

Figure 1: Cost-performance tradeoffs across agent-model configurations. The Pareto frontier (red dashed line) shows optimal tradeoffs: GPT 5.2 configurations offer the best cost-efficiency while Claude Opus 4.5 achieve the highest performance at 3–33× higher cost. Each point aggregates $n = 550$ tasks; model-level contrasts aggregate $n = 2{,}750$ per model, giving tight CIs of $\sim \pm 1.8$pp (App. F).

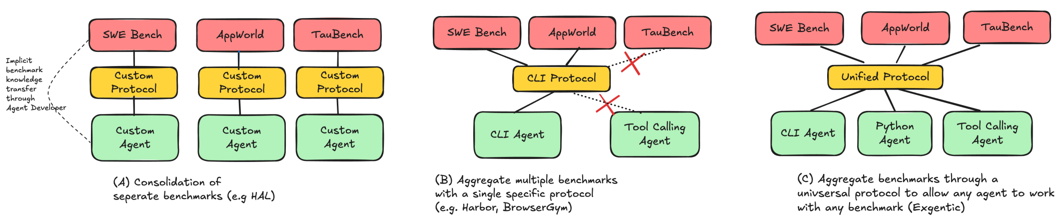

Figure 2: Evolution of Agentic Evaluation. (A) Collection of separate benchmarks, each requiring a custom agent or an agent with specific adaptation per benchmark (HAL) (B) Multiple benchmarks consolidated through a single protocol, such as CLI, or Web (C) Multiple benchmarks consolidated through a common protocol that can be adapted to any agent's protocol (Exgentic).

2 Unified Protocol Methodology

This work provides an evaluation solution for any general agent on any agentic benchmark, overcoming the common case of incompatibility between agent and benchmark protocols that either prevent evaluation (Fig. 2(B)), or require costly pairwise adaptation for each agent and benchmark (Fig. 2(A)). To address these limitations, we introduce a Unified Protocol that serves as a mediation layer between agents and benchmarks.

The Unified Protocol serves as a "narrow waist", adding a new agent (or benchmark) only needs adhering to it rather than to all benchmarks (agents). 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. This is what makes the protocol truly unified: if any semantic were missing, agents or benchmarks relying on it could not be represented, and the Unified Protocol would fail to be unified.

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 B).

1. **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*

Figure 3: Open General Agent Leaderboard is the first benchmark to consistently test general-agent architectures across key skills in diverse environments.

helps the user according to the policy provided below. Try to be helpful and always follow the policy.”. In addition, the first user utterance, such as *“Cancel my flight reservation AH3BDS”*, is passed to the agent separately as the first observation from the environment.

2. **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*. We note that the agent can use the context in different ways. For example, the agent can naively append it to the task or store it in a dedicated agent memory or document store for conditional retrieval.
3. **Actions**: *What can the agent do?*
   A set of environment actions. These actions constitute the complete set of operations the environment makes 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 for the airline domain, some actions are $\mathrm{cancel\_reservation}(reservation\_id)$ and $\mathrm{search\_direct\_flight}(origin, destination, date)$.

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. In this work we interface with five distinct protocols: tool-calling APIs (ReAct), MCP (OpenAI Solo), Python code generation (Smolagent), bash/CLI (SWE-Bench Verified-style execution), and conversational messaging (τ^2-Bench). While these protocols all accept multimodal inputs, the benchmarks we evaluate, widely adopted by frontier-model developers, are selected to stress *cross-domain capability* rather than multimodal processing; further discussion in Appendix H.

2.2 Running the Evaluation

Using the Unified Protocol, we manually wired six domain-specific benchmarks and five heterogeneous agent protocols into a unified evaluation pipeline. For each benchmark, we derive the agent-visible interface from a reference agent implementation, preserving the benchmark’s intended semantics; each adaptor then runs the original agent and benchmark as a black box. Exgentic, our harness implementing the Unified Protocol, orchestrates these adaptors and runs the 90 agent $\times$ model $\times$ benchmark configurations in isolated, reproducible sessions. Appendix C walks through one complete adaptation example (MINI-SWE AGENT→SWE-Bench Verified) and the orchestrator design.

3 Experimental Setup

We evaluate 5 agent architectures across 3 frontier LLMs (GPT 5.2, Claude Opus 4.5, Gemini 3 Pro) on 6 benchmarks (max 100 turns per task), yielding 90 configurations (100 tasks per benchmark, 50 for τ^2-Bench Airline). 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. Appendix B provides detailed descriptions of the benchmark adaptations to the Unified Protocol.

3.1 BENCHMARKS

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 500 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 , 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

ReAct We implement two ReAct-style (Yao et al., 2023) agents: a vanilla ReAct baseline using LiteLLM's tool-calling interface, and an extended version with tool shortlisting. 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 Smolagents 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 OpenAI's SDK v0.7.0 in solo mode with Model Context Protocol 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 general effectiveness 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 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 D details their presence across agents.

[1] Building agents with the Claude Agent SDK.

3.3 METRICS

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

4 RESULTS

Our main results address three central questions: (1) Do agents generalize across domains? (2) What drives agent performance, model quality or agent architectural design? (3) What architectural components enable cross-domain capabilities?

4.1 KEY LEADERBOARD FINDINGS

Our benchmark evaluation reveals clear performance hierarchies at the model, agent, and configuration levels. These findings provide practitioners with actionable guidance for system selection and deployment (see Appendix E for complete leaderboard results).

Top Configurations: The leaderboard (Table 1) is split between Claude Opus 4.5 and Gemini 3 based pairings, with Claude Opus 4.5 occupying the top three positions. No GPT 5.2 configuration appears in the top-10.

We assessed statistical significance using a pooled McNemar test. While the top configuration (utilizing OpenAI Solo and Claude Opus 4.5) did not significantly outperform the second-ranked configuration, it demonstrated a significant advantage over the third-ranked ($p < 0.01$) and all remaining configurations ($p < 0.001$). See Appendix F for detailed statistical analysis.

Model Performance: We compare models using their mean success rate across all agents and benchmarks, weighting τ^2-Bench subdomains equally (1/12 each) to balance benchmark representation. Claude Opus 4.5 ranks first with a success rate of 0.66, followed by Gemini 3 at 0.60, while GPT 5.2 underperforms at 0.40. Pairwise statistical tests over all (benchmark, task, agent) configurations confirm that these performance differences are significant ($p < 0.0001$). Claude Opus 4.5's superiority is consistent across nearly all benchmarks, whereas GPT 5.2's low aggregate performance is largely driven by failures in tool-rich environments.

Agent Performance: We compare agents by their mean success rate across all models and benchmarks (weighting τ^2-Bench subdomains as 1/12 each to balance benchmark representation). ReAct Short leads with 0.57, closely followed by OpenAI Solo (0.57), ReAct (0.55), Claude Code (0.54), and Smolagent (0.53). Using paired McNemar test, comparing results over each pair of agents on all (benchmark, task, model) combinations, we saw that these differences not statistically significant ($p > 0.1$).

However, agent performance are model-dependent: OpenAI Solo excels with Claude Opus 4.5 (0.73) but struggles on GPT 5.2 (0.39), while ReAct Short performs more consistently across models.

Notable Outliers: OpenAI Solo + Gemini 3 achieves the highest single-benchmark score (0.89 on τ^2-Bench-Telecom), while four GPT 5.2 configurations score 0.00 on AppWorld without tool shortlisting, as GPT 5.2 is limited to 128 tools while AppWorld exposes 468. This is a model limitation, not an integration artifact: Claude Opus 4.5 scores 0.65–0.80 on the same environment with no shortlisting, and adding shortlisting to ReAct+GPT 5.2 raises its AppWorld score from 0.00 to 0.33. The Unified Protocol pathway itself is not the bottleneck. The performance spread within models ranges from 11 percentage points (Claude Opus 4.5: best 0.73, worst 0.62) to 6 percentage points (Gemini 3: best 0.62, worst 0.56), indicating that agent choice matters significantly even for strong models.

[2]Model prices and context window.

4.2 No Single Agent Dominates Across Task Domains

Table 2 shows the best-performing agent-model configuration for each benchmark.

Table 2: Success rate (Score) of best agent-model configuration per benchmark. Top Score denotes the highest reported domain-specific agent performance on the original leaderboard (links are in App. E.2). Note: our results are on 100 randomly sampled benchmark instances, whereas the original leaderboard reports results on the full benchmark.

Benchmark	Best Configuration	Score	Top 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	OpenAI Solo + Gemini 3	0.89	0.98
AppWorld	Smolagent + Claude Opus 4.5	0.7	0.73

No single agent dominates: OpenAI Solo wins 4 benchmarks and ties on 1 (SWE-Bench Verified, τ^2-Bench-Airline, τ^2-Bench-Retail, τ^2-Bench-Telecom, and BrowseComp+ tie), demonstrating particular strength on structured API interaction tasks and code generation. Smolagent wins 1 benchmark and ties on 1 (AppWorld and BrowseComp+ tie), excelling on web navigation and multi-application environments.

4.3 Model Quality Drives Performance

Because our factorial design evaluates the same unmodified agents against a consistent set of models and benchmarks, it is the first to allow direct variance decomposition of model vs. agent contribution. We performed variance decomposition to isolate the relative contributions of model choice versus agent architecture. We compute variance explained as $\eta^2 = \text{Var}(\mathbb{E}[Y|X])/\text{Var}(Y)$, where Y is the task success rate and X is the grouping variable (model or agent). Model choice accounts for 28.2% of total success rate variance across all configurations, while agent architecture explains only 0.6%. The remaining 71.2% reflects task-level variance: differences in benchmark difficulty, task characteristics, and stochastic execution. Model quality is by far the strongest single factor, dominating agent architecture by more than 85-fold, a gap that is statistically decisive across our $\sim$8,250 observations.

4.4 Model Stability to Agent Architectures

Beyond average performance, model stability across different agent architectures is critical for practical agent development. A stable model allows developers to iterate on agent design without model-specific tuning; an unstable model requires careful co-design of the agent-model pairing. We measure stability as the standard deviation of scores across agent architectures for each model.

Claude Opus 4.5 exhibits the highest stability (Mean: 0.66, STD: 0.06), followed by GPT 5.2 (Mean: 0.40, STD: 0.071) and Gemini 3 (Mean: 0.59, STD: 0.09). Claude's standard deviation is slightly lower than Gemini 3's, indicating that agent performance with Claude varies minimally across architectural choices.

Stability has practical implications for agent development workflows. With Claude Opus 4.5, developers may focus on agent architecture without extensive model-specific optimization. With Gemini, agent-model co-design may becomes necessary, increasing development cost and reducing modularity. For practitioners prioritizing development efficiency and deployment flexibility, model stability may be as important as absolute performance.

4.5 Agent Components Effect

Several agent components discussed in Section 3.2.1 prove useful across agent implementations or models. Notably, the top three performing architectures (OpenAI Solo, Claude Code, and Smolagent) all employ a schema guard component: a mechanism that detects when an action with an invalid schema is invoked and allows the agent to correct itself. This highlights the potential value of such self-correction mechanisms across agents based on Claude Opus 4.5. Tool shortlisting, when added to a simple ReAct agent with tool calling, improves performance across all models in tool-rich environments. For GPT 5.2, shortlisting adds 5 percentage points overall, while for Claude Opus 4.5 the gain is more modest (1 percentage point) but comes with a substantial cost reduction of $1.97 on average. These results underscore the importance of documenting agent components, sharing implementation details, and conducting ablation studies as a primary means of advancing general agent development.

4.6 Cross-Benchmark Agent Stability

To assess whether agent performance generalizes across task types, we computed Spearman rank correlations between benchmark scores across all agent-model configurations. Results reveal moderate to strong positive correlations across most benchmark pairs: τ^2-Bench-Airline vs τ^2-Bench-Retail shows $+0.85$ (strong positive), SWE-Bench Verified vs τ^2-Bench-Telecom shows $+0.78$, and AppWorld vs τ^2-Bench-Retail shows $+0.75$. BrowseComp+ shows more moderate correlations with other benchmarks (0.32 to 0.74), suggesting it captures somewhat distinct capabilities while still following overall model quality trends.

The predominantly positive correlations reflect systematic model differences: GPT 5.2 underperforms across nearly all benchmarks (mean 0.40) while Claude Opus 4.5 excels broadly (mean 0.66). Within-model analysis shows that agent rankings vary by model (on Claude Opus 4.5, OpenAI Solo leads at 0.73; on GPT 5.2, ReAct Short leads at 0.46), so optimal agent choice depends on the underlying model.

4.7 Cost-Efficiency Tradeoffs

Cost-efficiency (score per dollar of inference cost) varies by 30$\times$ across configurations: GPT 5.2 configurations dominate the efficiency rankings (ReAct+GPT 5.2 at 2.41 score/$), while the best-performing overall, OpenAI Solo+Claude Opus 4.5 (0.73 average score), operates at 0.08 score/$. Figure 1 visualizes the Pareto frontier; Appendix E provides the full per-configuration breakdown (Table 6). There is no universal "best" choice: optimal selection depends on application-specific cost-performance requirements.

4.8 Failure Patterns and Agent Behavioral Differences

We examined whether failures are associated with increased number of interactions between the agent and the environment, resulting in higher computational cost, and whether this pattern is consistent across agents architectures.

To this end, we compared successful and failed runs at the task level, aggregating across backbone models for each benchmark and agent architecture[3].

For each benchmark and agent architecture, we report the percent increase in mean steps for failed runs relative to successful runs (positive means failures are longer). Table 3 shows this gap is positive in most settings, with the largest overheads on interaction-heavy benchmarks such as AppWorld and BrowseComp+ (e.g., ReAct is $+110.7\%$ on AppWorld). A few τ^2-Bench cells are near zero or negative, indicating some failures terminate early. The detailed interactions counts appear in App. E.3.

Benchmark-weighted averages are positive for every agent architecture (Claude Code 38.8%, OpenAI Solo 20.0%, Smolagent 26.4%, ReAct 54.4%, ReAct Short 45.1%), implying that failed runs generally consume more steps, and therefore more cost, than successful runs, amplifying the practical penalty of unreliability.

This pattern indicates that tasks that ultimately fail tend to take longer, whereas easier tasks complete more quickly. While this trend appears across all architectures, they differ in the magnitude of the effect. These differences suggest variations along more subtle dimensions than overall performance and cost (for example, how architectures allocate interaction budget, manage uncertainty, or recover from partial progress) that may be important when designing or selecting agent architectures.

[3] We excluded zero-step sessions and capped step counts at 50 to reduce the influence of long-tail outliers, which accounts to 7% of total runs.

Table 3: How much longer failed runs are than successful ones, measured by the percentage difference in number of interactions. Positive values mean failures take more interactions; negative values mean they take fewer.

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%

4.9 THE CURRENT STATE OF GENERAL-PURPOSE AGENTS

Four overarching themes emerge from our evaluation. First, **model quality creates cross-benchmark consistency**. No agent achieves consistently strong performance across all benchmarks, but strong positive correlations (0.75-0.85) across most benchmark pairs reflect systematic model differences. Claude Opus 4.5 excels broadly (mean 0.66), Gemini shows moderate performance (mean 0.60), and GPT 5.2 underperforms significantly (mean 0.40). Agent rankings vary within models, but model effects dominate overall patterns.

Second, **model quality dominates agent architecture**. Model choice explains 28.2% of score variance while agent architecture explains only 0.6%. The model-agent interaction effect (5.0%) exceeds the agent main effect by more than 4.5-fold, indicating that optimal agent selection depends heavily on the model. However, model differences, particularly GPT 5.2 failures on tool-rich environments, drive most performance variation. Agent architecture matters primarily for enabling model capabilities (e.g., ReAct Short for GPT 5.2) rather than as an independent performance driver.

Third, **practical deployment considerations (cost, tool scalability, component complexity) are not secondary concerns but fundamental constraints**. Tool shortlisting transforms GPT 5.2 from unusable to viable in tool-rich environments. Cost-efficiency varies by 30× across configurations. Sophisticated components like memory and planning correlate with gains but increase implementation complexity. These tradeoffs define the space of viable deployments rather than being post-hoc optimizations.

Fourth, **general-purpose agents are competitive with benchmark-specific heavily customized agents**. Our results show that across benchmarks, general agents largely match or exceed specialized systems (Tab. 2). Overall, although general agents have not yet been systematically pursued in full and still have substantial room to improve, these results establish general agents as a promising direction for future research and development.

Progress toward general-purpose agents requires addressing generalization explicitly through cross-benchmark evaluation. Improving single-benchmark performance does not yield generalizing agents.

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), each targeting specific domains such as software engineering (Jimenez et al., 2023; 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[4] models agents and benchmarks as interacting via A2A/MCP subsets, standardizing evaluation lifecycle components but leaving task semantics to individual benchmarks. CUBE (Lacoste et al., 2026) proposes a shared schema for unifying agent benchmarks. The Unified Protocol operates at the *integration* layer, mediating between agent protocols and benchmark interfaces via a canonical task/context/actions representation. 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.

6 Discussion

This work reframes general-agent evaluation as a first-class research target. A full factorial over unmodified agents $\times$ models $\times$ benchmarks is required to isolate agent contribution from model and benchmark effects. Our initial results demonstrate that agents can generalize across domains without domain-specific adaptation, matching or exceeding domain-specific performance across most benchmarks and establishing general agents as a viable alternative (Tab. 2). Our evaluation reveals promising opportunities for advancement: substantial performance headroom, domain variations suggesting architectural improvements, and clear cost-performance optimization targets. These findings point to exciting research directions: enhancing performance through better reasoning and planning, achieving cross-domain consistency, developing cost-effective solutions, and expanding to multimodal and safety-critical scenarios. The Open General Agent Leaderboard and Exgentic provide a foundation for systematic comparison and iterative progress toward truly capable general-purpose agents.

Limitations. Our evaluation is bounded by cost and scope: three frontier LLMs, five agent implementations, and six benchmarks, for 90 agent $\times$ model $\times$ benchmark configurations at $\sim$\$22K. Per-benchmark scores at $n = 50$–100 carry Wilson CI half-widths of $\pm$7–9.5pp, while aggregated and model-level comparisons use substantially more observations and remain highly significant (App. F). While the agent protocols we evaluate accept multimodal inputs, the benchmarks here are chosen to stress cross-domain capability rather than multimodal processing; extending to environments with continuous action spaces (e.g., pixel-level computer-use) is discussed in App. H.

Sustainability and adoption. Our approach is designed for incremental extension: new protocol adaptors require $\sim$200 lines of code, and agents or benchmarks that reuse a supported protocol can be integrated through it. Because the Unified Protocol operates at the integration layer, it composes naturally with existing infrastructure harnesses.

Impact Statement

The current research landscape for AI agents is fragmented by domain-specific benchmarks and communication protocols, which limit the development of general-purpose systems. This work introduces Exgentic and the Unified Protocol to bridge these gaps, enabling the first systematic evaluation of general agents across diverse environments. By establishing the Open General Agent Leaderboard, we provide the research community with a foundation for developing agents that transcend individual tasks and generalize across heterogeneous real-world settings. Our findings highlight that while model quality remains the primary driver of performance, standardized evaluation is essential for identifying the architectural components that enable scalable, cross-domain capabilities.

As with any public leaderboard, we acknowledge the risk of benchmark gaming and Goodhart's law: optimizing directly for Open General Agent Leaderboard scores may encourage overfitting to the specific benchmarks it aggregates rather than genuine cross-domain generalization. Yet benchmarks both measure progress and set direction for the field; without them, general-agent research cannot be tracked or focused. Agent-native evaluation partially mitigates this, since agents are not pre-fitted to a shared protocol. We encourage contributors to add benchmarks that stress unseen environments and to treat leaderboard position as one signal among many.

[4]AgentBeats

REFERENCES

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

Elron Bandel, Asaf Yehudai, Alexandre Lacoste, Avijit Ghosh, Graham Neubig, Margaret Mitchell, Michal Shmueli-Scheuer, and Leshem Choshen. Position: Agentic systems should be general. *SSRN Electronic Journal*, February 2026a. doi: 10.2139/ssrn.6176178. URL https://ssrn.com/abstract=6176178.

Elron Bandel, Asaf Yehudai, and Michal Shmueli-Scheuer. 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/.

Jonathan Bragg, Mike D'Arcy, Nishant Balepur, Dan Bareket, Bhavana Dalvi, Sergey Feldman, Dany Haddad, Jena D Hwang, Peter Jansen, Varsha Kishore, et al. Astabench: Rigorous benchmarking of ai agents with a scientific research suite. *arXiv preprint arXiv:2510.21652*, 2025.

Zijian Chen, Xueguang Ma, Shengyao Zhuang, Ping Nie, Kai Zou, Andrew Liu, Joshua Green, Kshama Patel, Ruoxi Meng, Mingyi Su, Sahel Sharifymoghaddam, Yanxi Li, Haoran Hong, Xinyu Shi, Xuye Liu, Nandan Thakur, Crystina Zhang, Luyu Gao, Wenhu Chen, and Jimmy Lin. Browsecomp-plus: A more fair and transparent evaluation benchmark of deep-research agent, 2025. URL https://arxiv.org/abs/2508.06600.

Thibault Le Sellier De Chezelles, Maxime Gasse, Alexandre Drouin, Massimo Caccia, Léo Boisvert, Megh Thakkar, Tom Marty, Rim Assouel, Sahar Omidi Shayegan, Lawrence Keunho Jang, Xing Han Lù, Ori Yoran, Dehan Kong, Frank F. Xu, Siva Reddy, Quentin Cappart, Graham Neubig, Ruslan Salakhutdinov, Nicolas Chapados, and Alexandre Lacoste. The browsergym ecosystem for web agent research, 2025. URL https://arxiv.org/abs/2412.05467.

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

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. Swe-bench: Can language models resolve real-world github issues? *arXiv preprint arXiv:2310.06770*, 2023.

Sayash Kapoor, Benedikt Stroebl, Peter Kirgis, Nitya Nadgir, Zachary S Siegel, Boyi Wei, Tianci Xue, Ziru Chen, Felix Chen, Saiteja Utpala, Franck Ndzomga, Dheeraj Oruganty, Sophie Luskin, Kangheng Liu, Botao Yu, Amit Arora, Dongyoon Hahm, Harsh Trivedi, Huan Sun, Juyong Lee, Tengjun Jin, Yifan Mai, Yifei Zhou, Yuxuan Zhu, Rishi Bommasani, Daniel Kang, Dawn Song, Peter Henderson, Yu Su, Percy Liang, and Arvind Narayanan. Holistic agent leaderboard: The missing infrastructure for ai agent evaluation, 2025. URL https://arxiv.org/abs/2510.11977.

Alexandre Lacoste, Nicolas Gontier, Oleh Shliazhko, Aman Jaiswal, Kusha Sareen, Shailesh Nanisetty, Joan Cabezas, Manuel Del Verme, Omar G. Younis, Simone Baratta, Matteo Avalle, Imene Kerboua, Xing Han Lù, Elron Bandel, Michal Shmueli-Scheuer, Asaf Yehudai, Leshem Choshen, Jonathan Lebensold, Sean Hughes, Massimo Caccia, Alexandre Drouin, Siva Reddy, Tao Yu, Yu Su, Graham Neubig, and Dawn Song. Cube: A standard for unifying agent benchmarks, 2026. URL https://arxiv.org/abs/2603.15798.

P. Langley. Crafting papers on machine learning. In Pat Langley (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. Agentbench: Evaluating llms as agents. *arXiv preprint arXiv:2308.03688*, 2023.

Sami Marreed, Alon Oved, Avi Yaeli, Segev Shlomov, Ido Levy, Offer Akrabi, Aviad Sela, Asaf Adi, and Nir Mashkif. Towards enterprise-ready computer using generalist agent, 2025. URL https://arxiv.org/abs/2503.01861.

Mike A Merrill, Alexander G Shaw, Nicholas Carlini, Boxuan Li, Harsh Raj, Ivan Bercovich, Lin Shi, Jeong Yeon Shin, Thomas Walshe, E Kelly Buchanan, et al. Terminal-bench: Benchmarking agents on hard, realistic tasks in command line interfaces. *arXiv preprint arXiv:2601.11868*, 2026.

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

Aymeric Roucher, Albert Villanova del Moral, Thomas Wolf, Leandro von Werra, and Erik Kaunismäki. ‘smolagents‘: a smol library to build great agentic systems. https://github.com/huggingface/smolagents, 2025.

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

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, 2024.

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

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. Tau-bench: A benchmark for tool-agent-user interaction in real-world domains. *arXiv preprint arXiv:2406.12045*, 2024.

Asaf Yehudai, Lilach Eden, Alan Li, Guy Uziel, Yilun Zhao, Roy Bar-Haim, Arman Cohan, and Michal Shmueli-Scheuer. Survey on evaluation of llm-based agents, 2025. URL https://arxiv.org/abs/2503.16416.

Yanzhao Zhang, Mingxin Li, Dingkun Long, Xin Zhang, Huan Lin, Baosong Yang, Pengjun Xie, An Yang, Dayiheng Liu, Junyang Lin, Fei Huang, and Jingren Zhou. Qwen3 embedding: Advancing text embedding and reranking through foundation models, 2025. URL https://arxiv.org/abs/2506.05176.

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

Shuyan Zhou, Frank F Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. Webarena: A realistic web environment for building autonomous agents. *arXiv preprint arXiv:2307.13854*, 2023.

A DETAILED BENCHMARK AGENT INTERACTION EXAMPLE

This section demonstrate 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 a *observation queue*.

Later, when the orchestrator calls

$$\text{action} = \mathrm{CodeAgentWrapper.\,react}(\text{observation}),$$

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, $\mathrm{react}(\cdot)$ 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 $\mathrm{react}(\cdot)$ call. The action is then returned to the orchestrator, which forwards it to the benchmark session, obtains the next observation, and calls $\mathrm{react}(\cdot)$ 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 $\mathrm{TauBenchBenchmark.\,start}()$, the list of available task names is retrieved from the τ^2-Bench codebase. When $\mathrm{TauBenchBenchmark.\,next\_session}()$ 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

$$\text{observation} = \mathrm{TauBenchBenchmark.\,step}(\text{action}),$$

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.

B BENCHMARK ADAPTATION

B.1 SWEBENCH 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`
- `finish`

B.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)`
- `submit(exact_answer: str, explanation: str, confidence: float)`
- `get_document(docid: str)`

B.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 (OVERALL 468)

- `finish`
- `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`
- `.... many more tools ...`
- `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`

B.4 TAU2BENCH 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. ......

ACTIONS

- `message`
- `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`

C 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.

C.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[5]. 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[6] 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.

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.

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

C.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 to 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 *message* to the corresponding *message* action.

C.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.

C.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 A.

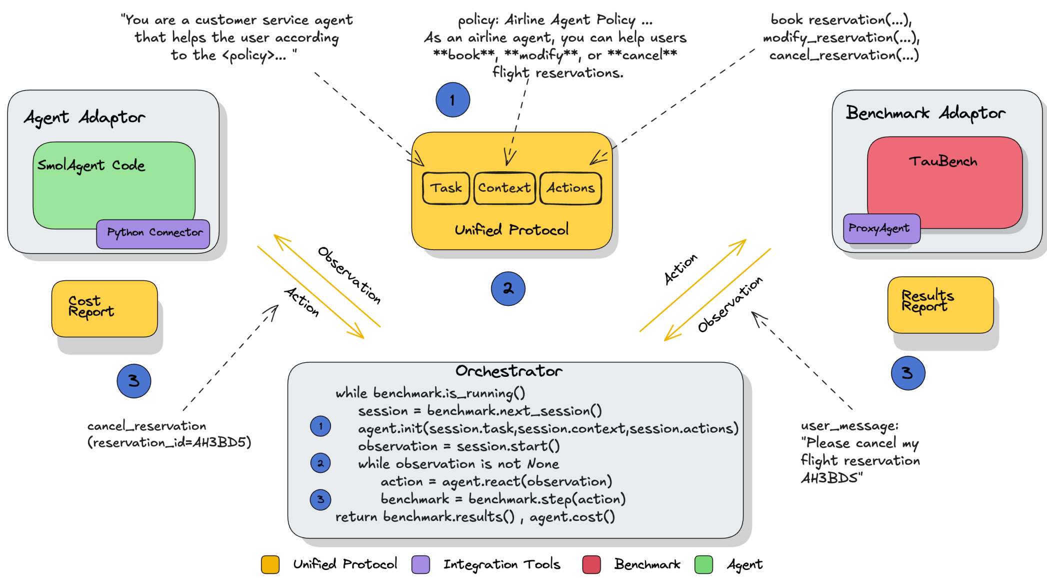

Figure 4: Exgentic architecture. Exgentic defines a unified protocol between agents and benchmarks. The Exgentic Orchestrator connects the agent and the benchmark, first passing the task definition and then mediates the observations and actions that are passed between the benchmark and the agent. Exgentic 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.

D Agent Components

We outline the key components and, in Table 4, analyze the components present in each agent.

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 MCP protocol, and Python-based agents receive runtime errors from the Python 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

Table 4: Architectural components of evaluated agents. ✓ denotes an explicit, modular component; $\overset{\times}{\checkmark}$ denotes an implicit or non-modular capability; ✗ denotes absence.

Agent	Execution Runtime	Tool Shortlisting	Tool Schema Guard	Communication Protocol	Memory	Planning
ReAct	✗	✗	✗	Tool-calling	$\overset{\times}{\checkmark}$	$\overset{\times}{\checkmark}$
ReAct Short	✗	✓	✗	Tool-calling	$\overset{\times}{\checkmark}$	$\overset{\times}{\checkmark}$
Smolagent	✓	✗	✓	Python-Functions	$\overset{\times}{\checkmark}$	$\overset{\times}{\checkmark}$
OpenAI Solo	✗	✗	✓	MCP	$\overset{\times}{\checkmark}$	$\overset{\times}{\checkmark}$
Claude Code	✓	✗	✓	MCP	✓	✓

Table 5: Agent-Model Configuration Leaderboard

Agent	Model	App	Browse	SWE	Airline	Retail	Telecom	Mean Score	Steps (avg)	Cost ($)
OpenAI Solo	Claude Opus 4.5	0.68	0.61	0.81	0.74	0.85	0.84	0.73	30.7	8.54
Claude Code	Claude Opus 4.5	0.66	0.53	0.74	0.66	0.83	0.76	0.67	31.7	8.03
Smolagent	Claude Opus 4.5	0.70	0.61	0.65	0.72	0.78	0.58	0.66	29.2	4.39
ReAct Short	Gemini 3	0.55	0.48	0.71	0.70	0.82	0.73	0.62	18.8	0.66
ReAct Short	Claude Opus 4.5	0.64	0.49	0.61	0.66	0.78	0.76	0.62	24.5	3.78
ReAct	Gemini 3	0.51	0.48	0.71	0.70	0.82	0.73	0.61	18.6	0.81
ReAct	Claude Opus 4.5	0.61	0.49	0.61	0.66	0.78	0.76	0.61	25.0	5.75
OpenAI Solo	Gemini 3	0.58	0.33	0.72	0.62	0.73	0.89	0.60	21.3	2.81
Claude Code	Gemini 3	0.36	0.51	0.67	0.70	0.78	0.69	0.57	29.0	2.47
Smolagent	Gemini 3	0.13	0.57	0.76	0.68	0.76	0.88	0.56	32.2	1.85
ReAct Short	GPT 5.2	0.22	0.46	0.57	0.54	0.73	0.54	0.46	12.3	0.26
ReAct	GPT 5.2	0.00	0.46	0.57	0.54	0.73	0.54	0.41	9.8	0.17
OpenAI Solo	GPT 5.2	0.00	0.48	0.55	0.50	0.54	0.53	0.39	11.3	0.19
Claude Code	GPT 5.2	0.00	0.43	0.58	0.48	0.51	0.55	0.38	10.7	0.38
Smolagent	GPT 5.2	0.07	0.26	0.53	0.60	0.68	0.71	0.38	22.2	0.36

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.

E Detailed Results

Table 1 presents the complete leaderboard results, including the average number of steps.

E.1 Cost-Efficiency per Configuration

Table 6: Most and least cost-efficient configurations per model. Efficiency is average score divided by average cost per task.

Configuration	Score	Cost/Task	Efficiency
ReAct + GPT 5.2	0.41	$0.17	2.41
Claude Code + GPT 5.2	0.38	$0.38	1.00
ReAct + Gemini 3	0.62	$0.66	0.93
OpenAI Solo + Gemini 3	0.60	$2.81	0.21
ReAct Short + Claude Opus 4.5	0.62	$3.78	0.16
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[7], BrowseComp+ and AppWorld are 0.80[8], and 0.73[9], respectively. τ^2-Bench Airline (0.73), Retail (0.86), and Telecom (0.98)[10].

E.3 Steps Counts

Table 7: Average steps per benchmark and architecture, split by successful vs. failed sessions; models are aggregated, 0-step sessions are excluded, and steps are capped at 50.

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**

F Statistical Significance

We assess the statistical significance of the benchmark results. 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.

For a single benchmark with n = 100 binary trials, the 95% Wilson 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 models 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: model-level rankings aggregate across 5 agents $\times$ 550 instances $= 2{,}750$ observations per model (giving a 95% CI half-width of approximately $\pm$1.8pp), and the η^2 variance decomposition in Section 4.3 uses all $\sim$8,250 observations across 15 configurations. At these sample sizes, the Opus-vs-GPT gap (26pp) 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.

To enhance statistical power when comparing benchmarks, we employ McNemar's test for pairwise analysis. This allows us to determine if one configuration significantly outperforms another by isolating performance discrepancies on identical tasks.

G Reproducibility Details

Table 8 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.

[7] SWE-Bench Verified

[8] BrowseComp+

[9] AppWorld

[10] τ^2-Bench

Parameter	**Value**
LLMs	GPT 5.2, Claude Opus 4.5, Gemini 3 Pro
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 Verified, τ^2-Bench (Airline, Retail, Telecom)
Instances per benchmark	100 (τ^2-Airline: 50); 550 total per agent-model
Total configurations	90 (5 agents $\times$ 3 models $\times$ 6 benchmarks)
Total evaluation cost	$\sim$\$22K

Table 8: Configuration shared across all runs. Per-benchmark prompts and tool description templates follow the protocol in Appendix B.

H LIMITATIONS

While Exgentic provides a clear methodology and reusable building blocks for adaptation, familiarity with these capabilities and addition development work is 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, more over 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. 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.