Terminal Agents Suffice for Enterprise Automation

Patrice Bechard*
ServiceNow

Orlando Marquez
ServiceNow

Emily Chen
ServiceNow

Jordan Skelton
ServiceNow

Sagar Davasam
ServiceNow

Srinivas Sunkara
ServiceNow

Vikas Yadav
ServiceNow

Sai Rajeswar
ServiceNow
Mila – Quebec AI Institute
Université de Montréal

Abstract

There has been growing interest in building agents that can interact with digital platforms to execute meaningful enterprise tasks autonomously. Among the approaches explored are tool-augmented agents built on abstractions such as Model Context Protocol (MCP) and web agents that operate through graphical interfaces. Yet, it remains unclear whether such complex agentic systems are necessary given their cost and operational overhead. We argue that a coding agent equipped only with a terminal and a filesystem can solve many enterprise tasks more effectively by interacting directly with platform APIs. We evaluate this hypothesis across diverse real-world systems and show that these low-level terminal agents match or outperform more complex agent architectures. Our findings suggest that simple programmatic interfaces, combined with strong foundation models, are sufficient for practical enterprise automation.

CCS Concepts

• **Computing methodologies** → **Artificial intelligence**; *Intelligent agents*; • **Software and its engineering** → Empirical software validation; • **Information systems** → Web services.

Keywords

LLM Agents, Enterprise Automation, CLI Agents, Tool Use Agents, Web Agents, Model Context Protocol, Terminal Agents

1 Introduction

Large Language Models (LLMs) have rapidly evolved from code completion assistants [5, 8] to agents capable of executing multi-step tasks across software systems [17, 20, 42]. In enterprise settings, this shift is particularly consequential. Rather than drafting text or answering questions, LLM-powered agents are increasingly expected to perceive system state, reason about business context, and perform actions that modify operational data [7, 16, 21]. These agents operate over long horizons and interact with production systems where errors can cascade across records, approvals, and downstream workflows [15]. Enterprise automation therefore introduces new challenges, requiring agents to reliably interact with complex platforms and operate under real-world constraints.

To address these challenges, two architectural directions have become especially prominent. GUI-driven agents operate through web interfaces, issuing low-level actions over DOM elements and screenshots [10, 30]. Tool-augmented agents [26, 29, 39] expose curated action schemas through frameworks like Model Context Protocol (MCP) [1], enabling models to invoke predefined operations. Although these approaches differ in implementation, they share a common design choice: introducing structured abstractions between the model and the underlying platform, rather than enabling direct interaction.

Abstractions, however, come with tradeoffs. GUI agents must reason over long, brittle action chains that are sensitive to interface changes [28], while curated tool registries simplify invocation at the cost of restricting expressivity to predefined operations and compositions [34]. Recent generalist code agents such as Claude Code [2] and OpenClaw [32] demonstrate that strong performance on complex, real-world tasks can emerge without heavy, pre-curated abstraction layers by operating directly over programmable interfaces such as command-line environments and APIs. Modern enterprise platforms already expose similarly expressive APIs for querying data, updating records, and performing complex operations programmatically; in such settings, additional wrappers may reduce flexibility rather than enhance it, trading generality for structure without clear necessity.

In this work, we empirically test whether additional abstraction layers are necessary when stable APIs are available. We construct realistic enterprise automation benchmarks across multiple production-grade platforms and compare three paradigms: GUI-driven agents, tool-augmented agents, and minimal terminal-based coding agents that operate through direct API interaction. Across diverse workflows, **minimal terminal agents match or outperform more complex architectures despite their simplicity, while maintaining competitive efficiency**. These findings challenge the prevailing assumption that increasingly sophisticated agent stacks are required for enterprise automation, suggesting instead that strong foundation models combined with direct programmatic interfaces may suffice for a broad class of real-world tasks. Our main contributions are as follows:

- We show that simple terminal agents operating through direct API interaction are both effective and efficient for enterprise automation, outperforming MCP-based tool-augmented agents and matching or exceeding web-agent performance at substantially lower cost, based on a systematic evaluation of frontier LLMs across these three paradigms on diverse real-world enterprise tasks.
- We introduce a unified benchmark spanning multiple production platforms, including verified evaluation environments and datasets capturing realistic enterprise tasks.
- We further study practical extensions to terminal agents, including filesystem-based documentation access and self-created reusable skills.

*Corresponding author. patrice.bechard@servicenow.com

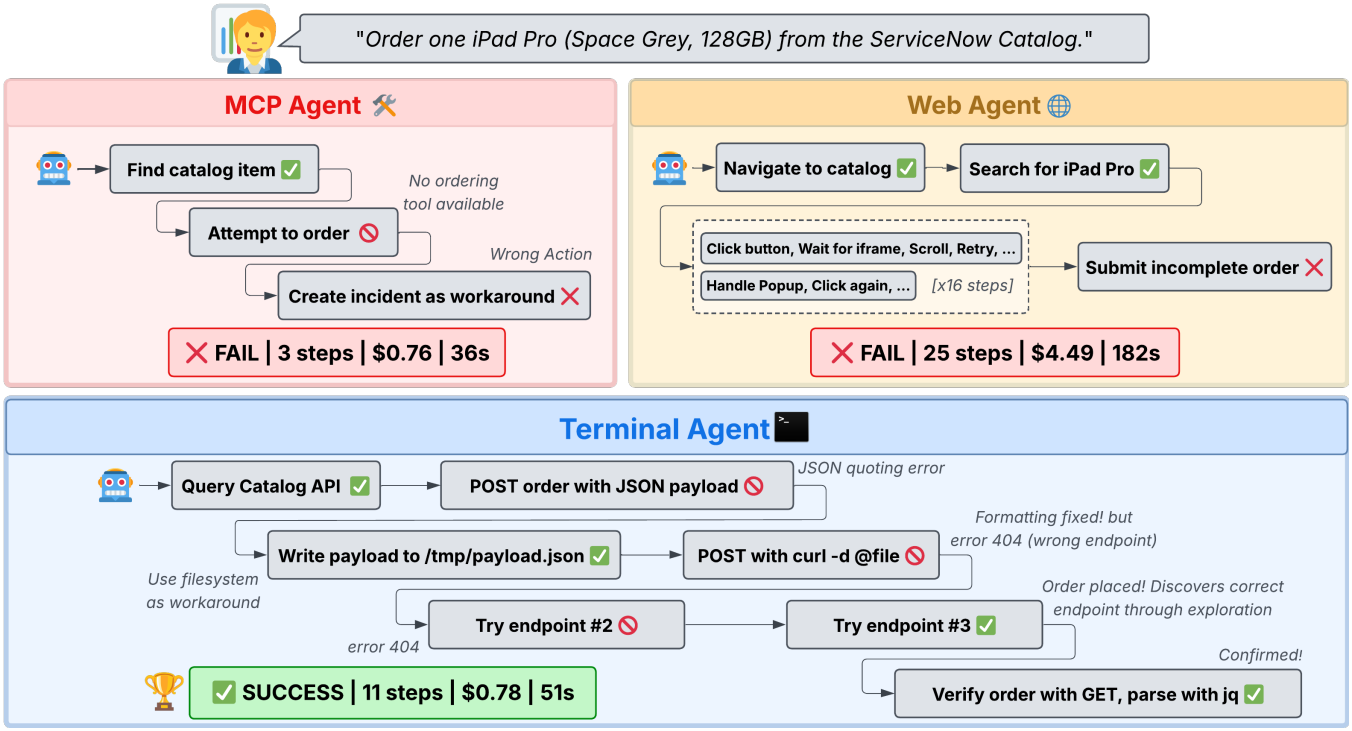

Figure 1: Execution traces of agents ordering an iPad Pro. (Top-left) The MCP agent identifies the catalog item but cannot proceed without an ordering tool. It falls back to creating a support ticket and fails. (Top-right) The web agent reaches the catalog page but becomes confused within the iframe-based UI, leading to a long, costly trajectory that fails to complete the order. (Bottom) The terminal agent encounters JSON quoting errors when constructing the request payload and 404 responses from incorrect API endpoints, but recovers by writing the payload to a temporary file and exploring alternative endpoints. It completes the task at an order of magnitude lower cost than the web agent, demonstrating the flexibility, resilience, and efficiency of direct terminal-based interaction.

Resources including evaluation environments, datasets, and code will be released upon acceptance to facilitate reproducibility and further research on enterprise agent systems.

2 Related Work

Web and GUI agents in enterprise benchmarks. A growing body of work studies agents that complete tasks through web or GUI interaction, translating natural-language instructions into sequences of clicks, typing, and navigation [10, 42]. WebArena [42] introduced a realistic and reproducible web environment and showed that even strong agents remain far below human performance on end-to-end success, particularly on tasks requiring long interaction chains. In enterprise settings, WorkArena [11] focuses on knowledge-work tasks on ServiceNow, and follow-up work [7] extends this setting to hundreds of compositional workflows, revealing planning, reasoning, and retrieval as key bottlenecks for current web-agent designs. Complementary benchmarks explore other enterprise platforms, including SCUBA for Salesforce CRM workflows [9] and TheAgentCompany for agents operating across multiple enterprise tools and services [36].

API-first and coding-agent approaches. A parallel line of work explores programmatic interaction as an alternative to GUI control, arguing that APIs and executable code provide more reliable and compact interfaces than step-by-step UI navigation. Beyond Browsing [31] evaluates this idea on WebArena by augmenting agents with API access, showing that API-enabled agents outperform browsing-only agents, while hybrid agents combining APIs and browsing perform best. AXIS [19] reaches similar conclusions in desktop environments, demonstrating that API-first "skills" can significantly reduce task completion time. Related work on coding agents treats executable code as the primary interface to the environment. CodeAct [34] frames agent actions as executable code, enabling direct interaction with external systems. AppWorld [33] introduces interactive coding environments with programmatic evaluation that verifies both intended state changes and unintended side effects. Complementary work such as Gorilla [27] shows that grounding models in retrieved API documentation improves reliability when interacting with evolving APIs.

Externalizing context through environment interaction. Enterprise automation tasks require agents to operate over information that exceeds the limits of a single prompt, motivating approaches that externalize context into the environment. Recursive Language Models (RLMs) [40] frame long-context processing as interaction with external tools through a REPL-style loop,

while software-oriented agents such as SWE-agent [38] and OpenHands [35] expose the filesystem and execution environment as part of the agent state. In parallel, work on memory and experience accumulation explores how agents can persist and reuse information across tasks [22, 25, 37, 41]. Our work adopts this perspective, using the filesystem and APIs as a simple mechanism for managing context and enabling reusable skills in enterprise automation tasks.

3 StarShell

We investigate whether terminal-based coding agents suffice for enterprise automation when APIs are available. To test this hypothesis, we compare three agent paradigms that differ only in their interaction modality: (1) GUI-driven web agents operating through browser interfaces, (2) tool-augmented agents accessing curated APIs via Model Context Protocol (MCP), and (3) minimal terminal agents interacting directly with platform APIs. All agents use the same LLM backbone and are evaluated on identical enterprise benchmarks to isolate the effect of the interaction paradigm.

3.1 Agent Interaction Paradigms

Tool-augmented agents operate through a curated set of API tools exposed via MCP servers, corresponding to operations such as listing records, creating entries, or updating fields. The agent selects a tool and provides arguments for invocation. This abstraction simplifies execution by compressing complex interactions into high-level operations, but constrains the agent to the functionality exposed by the tool registry.

Web agents operate through graphical interfaces, observing the rendered UI and issuing low-level actions such as clicking, typing, or navigating pages. In our implementation, web agents operate through the official Playwright MCP server [1], which exposes browser actions along with DOM elements and screenshots.

Our primary focus is a **simple coding agent that operates through a terminal and filesystem**. Instead of invoking predefined tools, it writes and executes code to interact directly with platform APIs, enabling flexible API interaction, data manipulation, and exploration. This pattern mirrors modern coding agents such as Claude Code and OpenClaw, where the model operates in a REPL-style loop of reasoning, execution, and environment inspection.

3.2 StarShell: A Terminal Enterprise Agent

We implement this terminal agent as StarShell, a minimal coding-agent environment for enterprise automation tasks. The agent operates through two primary interfaces: a terminal for executing commands and a filesystem for storing artifacts.

For each task, the agent receives the task description and execution context, then iteratively generates commands or code snippets to run. Typical actions include querying platform APIs, filtering results, updating records, or generating scripts to automate repetitive operations. API responses and execution outputs are returned to the model as observations, enabling iterative reasoning and correction.

The filesystem provides persistent task state. The agent can read documentation, cache intermediate results, and persist reusable "skills" such as scripts or notes capturing previously discovered solutions.

Unlike tool-based agents, StarShell does not rely on predefined action schemas. Instead, it discovers platform capabilities dynamically by reading documentation or inspecting API responses. This design allows the agent to compose operations that may not be represented in curated tool registries.

3.3 Enterprise Benchmark Environments

We evaluate agents across three enterprise platforms: ServiceNow,[2] GitLab,[3] and ERPNext,[4] which represent common categories of enterprise software: IT service management, software development lifecycle management, and enterprise resource planning. Table 1 summarizes the benchmark statistics.

Table 1: Summary statistics of the evaluation benchmark across three enterprise platforms.

	ServiceNow	GitLab	ERPNext
Samples	330	192	207
MCP Tools	93	107	7
Doc. Pages	61k	2.65k	5.41k

Each benchmark consists of natural-language tasks that require agents to inspect system state, retrieve information, and perform actions that modify platform records. The tasks range from simple record queries to multi-step workflows involving filtering, conditional updates, and reasoning across multiple objects. An exhaustive list of task types for each environment is provided in Appendix E.

For ServiceNow and GitLab, we build on tasks introduced in prior work [11, 42], but adapt the evaluation pipelines to use programmatic verification against the live platform rather than hardcoded values or browser-script checks. For ERPNext, we construct a new benchmark of enterprise workflows spanning multiple record types and task complexities. Details are provided in Appendix B.

All environments are deployed in containerized instances. We reuse the GitLab container setup from El Hattami et al. [12] and build custom environments for ServiceNow and ERPNext. For each platform, we provide (1) a sandbox environment with isolated data, (2) API access for programmatic interaction, (3) optional MCP tool registries exposing curated platform operations, and (4) a local documentation corpus accessible through the filesystem. This documentation is obtained by scraping official docs or using repository sources when available, and converting them to markdown for standardized access. We selected widely used open-source MCP servers implementations based on public availability, platform coverage, and community adoption. We do not further customize or expand these tool registries, as our goal is to evaluate practical MCP-based agents in realistic off-the-shelf conditions rather than through platform-specific tool engineering.

[1] https://github.com/microsoft/playwright-mcp

[2] Docs: https://www.servicenow.com/docs/; MCP: https://github.com/echelon-ai-labs/servicenow-mcp

[3] Docs: https://docs.gitlab.com/; MCP: https://github.com/zereight/gitlab-mcp

[4] Docs: https://docs.erpnext.com/; MCP: https://github.com/rakeshgangwar/erpnext-mcp-server

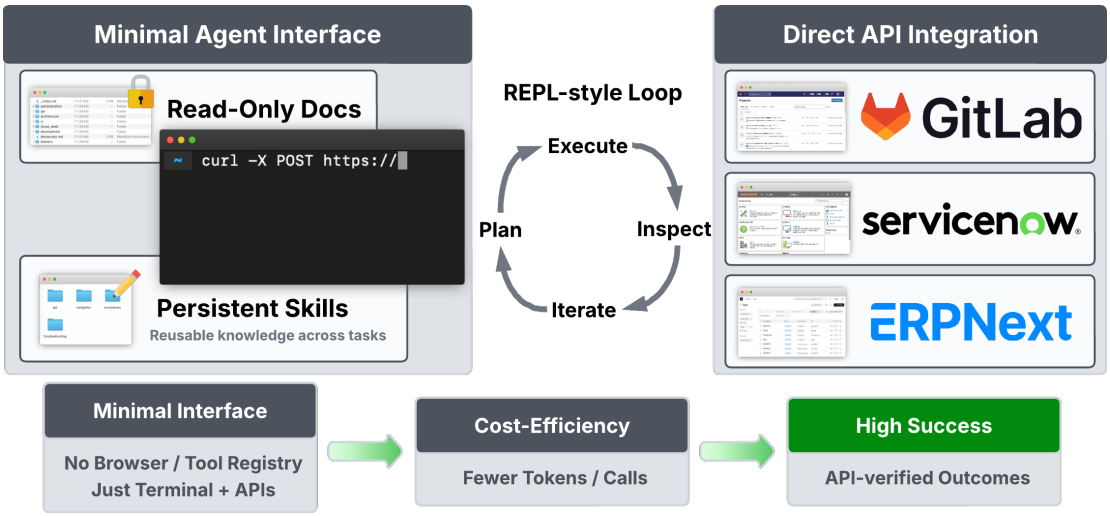

Figure 2: Overview of StarShell: a minimal terminal agent for enterprise automation. The agent operates through a terminal and filesystem, optionally using documentation and persistent skills to discover and invoke APIs directly on enterprise platforms (e.g., GitLab, ServiceNow, ERPNext), without relying on GUI interaction or pre-defined tool registries.

3.4 Experimental Setup

Agent implementations. All agents are implemented using the OpenAI Agents SDK [24]. Web agents use a Playwright MCP server that exposes browser actions and observations. Tool-augmented agents access curated platform APIs via MCP servers for ServiceNow, GitLab, and ERPNext. Terminal agents operate in a sandboxed environment with terminal and filesystem access. The tools exposed by the platform-specific MCP servers are listed in Appendix F; those exposed by the Playwright MCP server are listed in Appendix G.

We evaluate multiple frontier LLM backbones, including Claude Sonnet 4.6 [4], Claude Opus 4.6 [3], GPT-5.4 Thinking [23], and Gemini 3.1 Pro [14]. Within each comparison, all agent paradigms share the same backbone to ensure fairness.

Experimental design. Our experiments proceed in two stages. We first compare the three interaction paradigms (web agents, tool-augmented agents, and terminal agents) under a minimal configuration without documentation access, or reusable skills. This isolates the effect of the interaction interface itself. We then introduce additional capability modules, such as documentation access and skill persistence, and measure their contribution through controlled ablations.

For each task, the agent receives a natural-language task description and interacts with the platform until the task is completed or a predefined execution limit is reached. Tasks are executed in isolated environments to avoid cross-task interference.

Metrics. Our primary metric is **success rate (SR)**, defined as the percentage of tasks successfully completed according to task-specific verifications over the resulting system state. This metric captures overall task effectiveness. Because task outcomes are binary, we estimate standard errors for SR using the sample-proportion estimator. In our result tables, we highlight scores that fall within one standard error of the best-performing method.

To evaluate efficiency, we report **inference cost**, computed from the token usage of the underlying language model. Cost provides a consistent and model-agnostic measure of efficiency that is less sensitive to environment-specific latency than wall-clock time.

We also report two auxiliary metrics in Appendix D: tool calls and wall-clock time. Tool calls capture interaction efficiency but not reasoning cost, while wall-clock time depends on environment and infrastructure latency. We therefore use cost as the primary efficiency metric.

4 Experiments

4.1 Comparing types of agents

Table 2 presents results comparing different agent types on tasks from three enterprise environments using 4 different base LLMs.

MCP agents are limited by the available tools. Across all models, MCP agents achieve the lowest success rates, particularly on ServiceNow (11.5–16.7%), where, despite access to 93 tools, many tasks require actions beyond what the available endpoints support. On GitLab (107 tools), performance improves to 45–49% but remains well below other agent types. Paradoxically, the platform where MCP agents perform closest to their counterparts is ERPNext, which exposes only 7 tools. This suggests that a small set of well-designed, general-purpose tools (ERPNext's API provides generic CRUD operations over all document types) can be more effective than a large catalog of narrow, specialized endpoints that may not cover the specific operations a task requires. MCP agents are consistently the cheapest, confirming that their bottleneck is tool coverage rather than efficiency. We investigate this further in Appendix A.1, where we restrict the evaluation to the subset of tasks that are feasible for all three paradigms. Even on this reduced set, MCP agents underperform due to rigid tool interfaces that limit which fields can be set and which query patterns can be expressed.

Web agents offer more flexibility at a higher cost. By interacting with the platform through a browser, web agents can in principle perform any action a human user would, and this flexibility translates to strong success rates: they achieve the highest or

Table 2: Main results across agent interaction paradigms. Success rate (SR, ↑) and average cost per task (↓) for MCP, web, and terminal agents on three enterprise platforms with four backbone LLMs. Bold indicates the best SR and lowest cost within each platform–model group.

Agent	ServiceNow (330)		GitLab (192)		ERPNext (207)		Overall (729)	
	SR (%)	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*
Claude Sonnet 4.6								
MCP	11.5	**0.76**	45.2	0.48	55.6	**0.14**	32.9	**0.51**
Web	**72.4**	4.49	**82.9**	0.88	61.8	3.63	**72.2**	3.29
Terminal	**73.6**	**0.78**	76.5	**0.28**	**67.6**	0.46	**72.7**	0.56
Claude Opus 4.6								
MCP	16.1	**0.66**	46.8	0.90	68.9	**0.17**	39.2	**0.58**
Web	**77.6**	4.21	**81.9**	0.85	**81.6**	6.49	**79.9**	3.97
Terminal	**79.1**	1.94	**80.2**	**0.50**	76.8	0.72	**78.7**	1.22
GPT-5.4 Thinking (Medium)								
MCP	18.5	**0.14**	47.9	0.40	62.8	**0.21**	38.8	0.23
Web	69.4	0.54	**81.4**	0.17	**72.5**	0.51	**73.4**	0.43
Terminal	**77.0**	0.20	71.3	**0.13**	70.0	0.24	**73.5**	**0.19**
Gemini 3.1 Pro								
MCP	14.2	**0.10**	48.9	0.15	62.8	**0.07**	37.1	**0.11**
Web	62.1	0.68	**84.6**	0.22	65.2	1.13	68.9	0.69
Terminal	**78.5**	**0.10**	79.8	**0.06**	**73.9**	0.10	**77.5**	**0.09**

tied-highest accuracy in 8 of the 12 platform–model combinations. However, this comes at a substantial cost premium. On ServiceNow, web agents cost 4–6× more than terminal agents across Claude models, as each interaction requires processing large accessibility trees and screenshots. The overhead is especially pronounced for Opus 4.6 on ERPNext, where web agents cost $6.49 per task compared to $0.72 for CLI: a 9× difference with only a modest accuracy gain (81.6% vs. 76.8%).

Terminal agents offer the best cost–performance tradeoff. Terminal agents match or exceed web agent accuracy in 7 of 12 platform–model combinations while consistently costing less, often by a factor of 5 and more. On ServiceNow, terminal-based coding agents achieve the highest success rate for all 4 models, and with Gemini 3.1 Pro the terminal agent reaches 77.5% averaged across all environments at just $0.09 per task, the most cost-efficient configuration in the table. Their advantage stems from programmatic interaction through shell commands and API calls, which avoids the overhead of UI rendering while retaining the flexibility to compose arbitrary operations that MCP tools may not expose. We investigate where terminal agents still fail in Section 5.1.

4.2 Parametric knowledge vs. documentation

We investigate whether terminal agents benefit from access to official platform documentation or whether they can rely on their parametric knowledge and direct API interaction. Results are presented in Table 3.

Overall, documentation does not provide a clear benefit: both configurations achieve comparable success rates across both models we evaluated. With Sonnet 4.6, the agent without documentation access is slightly better overall (72.7% vs. 71.8%), while with Opus 4.6 the results are nearly identical (78.7% vs. 79.2%). This suggests that agents can often operate effectively without external documentation, either because relevant knowledge is internalized or because the APIs are sufficiently discoverable at inference time.

The aggregate numbers, however, mask divergent effects across platforms. On ServiceNow, documentation hurts with Sonnet (−6.3%) and adds cost across both models, as agents spend a significant fraction of their tool call budget retrieving and reading documentation rather than executing the task. On ERPNext, documentation helps with Sonnet (+4.7%) at virtually no additional cost, indicating that the agent can integrate the retrieved information efficiently. On GitLab, documentation has no effect on accuracy but roughly doubles the cost, suggesting the agent reads documentation without benefiting from it. We analyze why documentation helps on some platforms and hurts on others in Section 5.2.

4.3 Access to self-generated Skills

We investigate whether terminal agents can effectively "learn on the job." As agents perform tasks over time, we allow them to store troubleshooting strategies, workflows, and other useful notes in a persistent "skills" directory, which they may organize freely. We sequentially go over the full dataset once and compare the cumulative number of task completed and cumulative cost of agents equipped with and without these skills over time. Results can be seen in Figure 3. Both configurations use Sonnet 4.6.

Persistent memory improves success rate across all three platforms, though the magnitude varies. The largest gain is on ERPNext (+5.8pp), where tasks frequently involve non-obvious field

Table 3: Effect of documentation access on terminal agents. Success rate and cost per task for terminal agents with and without access to official documentation. Bold indicates the best SR and lowest cost within each platform–model group.

Agent	ServiceNow (330)		GitLab (192)		ERPNext (207)		Overall (729)	
	SR (%)	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*
Claude Sonnet 4.6								
No Docs	**73.6**	**0.78**	76.5	**0.28**	67.6	**0.46**	**72.7**	**0.56**
With Docs	67.3	1.10	**79.1**	0.48	**72.3**	**0.47**	**71.8**	0.76
Claude Opus 4.6								
No Docs	79.1	1.94	**80.2**	**0.50**	**76.8**	**0.72**	**78.7**	**1.22**
With Docs	**81.2**	**1.60**	**78.7**	0.71	**76.3**	0.99	**79.2**	**1.19**

names and multi-step dependency chains that benefit from recorded procedures. ServiceNow improves by 3.6pp, while GitLab shows only a marginal gain (+1.6pp), consistent with its tasks being more straightforward API calls that the model can handle from parametric knowledge alone.

Beyond accuracy, the more striking effect is on cost. On ServiceNow, the agent with memory costs 43.7% less per task on average ($0.44 vs. $0.78), and on ERPNext the reduction is 16.8%. One might expect memory accumulation to act as an investment: the agent incurs a small upfront cost to build reusable skills, which can be amortized over subsequent tasks. In practice, this overhead is modest. The agent checks the skills directory in 1–2 calls and writes new entries only when encountering genuinely novel information, so the maintenance cost is small relative to the savings from avoiding redundant API exploration. However, these benefits are not uniform across platforms.

We also show how memory grows over time. On ServiceNow and ERPNext, the skills directory grows rapidly during the first 1/3 of tasks as the agent encounters new task types and records working procedures, then plateaus once most patterns have been observed. On GitLab, memory size remains minimal throughout the run: the agent writes skills only three times in total. This contrast suggests that the value of persistent memory depends on how unfamiliar the platform's API is to the base model: ERPNext, a less common platform, benefits the most, while GitLab's well-documented REST API offers little that the model does not already know. This conclusion is also consistent with what was found in Section 4.2. We discuss the types of skills that agents generate in Section 5.3. Full results obtained for all experiments are reported in Appendix D. Additional experiments can be found in Appendix A.

5 Analysis and Discussion

In this section, we perform error analysis to get a better qualitative understanding of how terminal agents operate. We build custom tooling to help us explore traces and compare runs (see Appendix H).

5.1 Where do terminal agents still fail?

5.1.1 Failure is not driven by tool unreliability. To understand failure modes, we analyze the traces from our ServiceNow baseline evaluation for Claude Sonnet 4.6 (Section 4.1). In this setting, nearly all tool calls (97.3%) are `curl` commands issued against the ServiceNow REST API, so tool call failures correspond almost entirely to API interaction errors. Table 7 in Appendix C presents an overview of the different tool call outcomes along with examples.

Figure 4 shows the distribution of tool calls per task. The histograms for successful and failed tasks are broadly similar, suggesting that task difficulty (not sheer volume of interaction) drives failure. However, two differences stand out: failed tasks show a heavier concentration near zero, indicating that some agents fail early without meaningful progress, and a slight bump above 30 tool calls, suggesting agents occasionally get stuck in unproductive loops before hitting the turn limit or giving up.

Figure 5 breaks down error types as a fraction of total tool calls in each group. The distributions are strikingly similar: successful

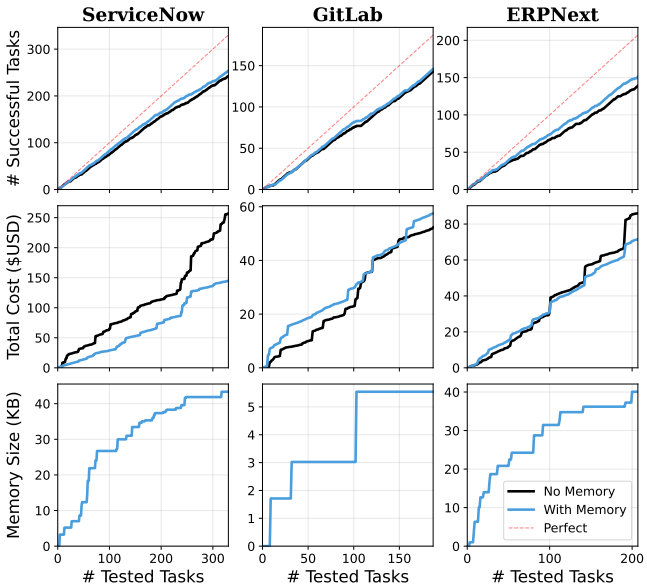

Figure 3: Skills accumulation over sequential tasks. Top: cumulative number of successful tasks. Middle: cumulative cost ($USD). Bottom: skills directory size (KB). The agent with memory (blue) accumulates reusable procedures; the baseline (black) starts fresh every time.

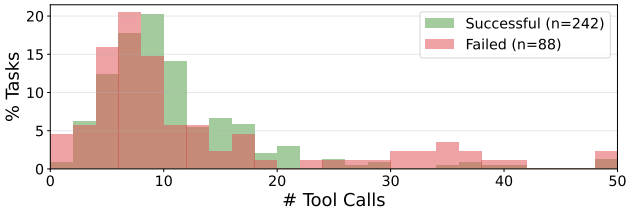

Figure 4: Distribution of tool calls per task for successful and failed tasks. Tasks exceeding 50 calls are capped at 50.

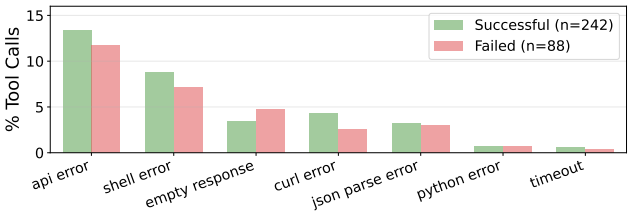

Figure 5: API error type breakdown as a percentage of total tool calls, comparing successful and failed tasks.

and failed tasks encounter comparable rates of API errors, `curl` failures, and shell errors. This indicates that the agent is generally able to recover from individual tool call failures, and that task-level failure is not simply caused by unreliable tools but by the agent's inability to make progress on the underlying task.

5.1.2 Limitations of the terminal interaction paradigm. While terminal agents achieve the best cost–performance tradeoff, they can fail on tasks requiring interaction with the platform state, which in some cases is only accessible through the web UI.

Some platform operations are inherently tied to the browser session and have no API equivalent. A clear example is ServiceNow's *impersonation* feature, which allows an administrator to act as another user. The terminal agent discovers what appears to be the correct endpoint (`/api/now/ui/user/impersonate`) and receives an HTTP 200 response, but the impersonation does not take effect: it requires a browser-session cookie exchange that the API call alone cannot perform. The web agent accomplishes the same task in two steps by clicking the user menu and selecting the target user.

Tasks that require reading rendered UI elements, such as extracting values from a chart, are fundamentally inaccessible to terminal agents. In these cases, the terminal agent queries the underlying tables and recomputes the aggregations, but the rendered output may differ due to rounding, formatting, or chart-specific display logic. The web agent, by contrast, can directly observe the rendered state.

The most significant limitation arises with tasks that involve interacting with complex UI interfaces, such as drag-and-drop workflow editors. Although it is not part of the tasks used for benchmarking, consider a task such as: *"Create a workflow triggered daily that looks up all high priority incidents assigned to Fred Luddy and send him an email with the details."* A web agent can complete this through standard UI interactions in Flow Designer. Although recent work shows that such workflows can be generated programmatically [6], this capability is not exposed through the platform's public APIs. As a result, terminal agents have no reliable way to perform this task.

These limitations suggest that combining programmatic and browser-based tools may be a promising direction, allowing agents to select the most effective interaction mode for each subtask. We explore this approach with a hybrid agent in Appendix A.3.

5.2 How do terminal agents leverage documentation?

Table 3 shows that documentation can either help or hinder terminal agents. To better understand this effect, we compare runs from Claude Sonnet 4.6 with and without access to the official product documentation for both ServiceNow and ERPNext. We find that documentation can be either beneficial or detrimental depending on how well its structure matches the agent's needs.

ServiceNow's documentation is primarily written for users performing tasks through the web UI, and its API pages are *reference-oriented*: they catalog the full API surface without indicating which endpoints are most appropriate for common operations. Agents reading these pages adopted unnecessarily complex strategies and spent a large fraction of their tool call budget on retrieval rather than task execution (Appendix J.1). ERPNext's documentation is *task-oriented*: each page describes a specific entity, its fields, and how to create or manage it, mapping directly to the API calls agents need to construct. This allowed agents to quickly find critical information, such as non-obvious doctype names, that cannot be discovered through schema exploration alone (Appendix J.2).

These findings suggest that documentation must be structured for its consumer. Human-oriented reference documentation can actively mislead agents, while concise, task-oriented content provides effective guidance. This motivates the use of *Skills*, structured procedures written by agents themselves or by humans, as a complement to static documentation [18].

5.3 What skills do terminal agents create?

We examine the contents of the skills directories accumulated during evaluation to understand what types of knowledge agents record and how this varies across platforms. Across all three platforms, the agent created 38 skill files: 11 for ServiceNow, 2 for GitLab, and 25 for ERPNext. Every skill was marked as `verified`, indicating the agent confirmed each procedure on at least one subsequent task.

Skills are overwhelmingly procedural, presenting step-by-step recipes for creating or modifying records via the platform API. However, the bulk of each skill's content is not the procedure itself, which is typically a single API call, but rather the *pitfalls* and *field mappings* embedded within it. These capture non-obvious knowledge discovered through trial and error: mappings between UI labels and API fields, valid field values, which endpoints exist, and workarounds for issues such as shell quoting.

The nature of skills varies markedly across platforms. On **ServiceNow**, skills are larger and more encyclopedic (3.9 KB on average), accumulating field mappings, identifiers, and query patterns across tasks. Much of their content captures non-obvious correspondences

between labels from the UI and field names from the API, as well as other platform-specific quirks.

On **ERPNext**, skills are more numerous but smaller (25 files averaging 1.8 KB), each focused on a single entity type. While the API structure is uniform, skills record subtle schema inconsistencies and side effects that can lead to silent failures, such as fields like `so_required=1` having inverted semantics (allowing invoices *without* sales orders).

On **GitLab**, only two skill files are created, both capturing non-obvious API conventions: the undocumented template name `plainhtml` for project creation and the access level mapping for member invitations. This scarcity suggests stronger parametric knowledge of the platform, as most tasks succeed without requiring new skills.

A recurring pattern across platforms is the documentation of **shell quoting issues** with `eval curl` and JSON payloads. Skills consistently warn against inline JSON and recommend using temporary files or scripts instead. This infrastructure-level lesson illustrates how skills capture knowledge that generalizes beyond individual tasks. An example of how agents can leverage skills to recall previously discovered solutions can be found in Appendix J.3.

6 Conclusion

This work examined whether sophisticated agent stacks are necessary for practical enterprise automation. Across realistic benchmarks spanning multiple production-grade platforms, we find that minimal coding agents that operate through a terminal and filesystem and interact directly with platform APIs match or outperform more complex architectures, such as GUI-driven web agents and MCP-based tool-augmented agents. Despite their simplicity, these agents achieve competitive efficiency while avoiding the brittleness of GUI interaction and the expressivity constraints of predefined tool schemas. More broadly, our results suggest that enterprise automation may benefit more from exposing stable programmable interfaces than from introducing additional abstraction layers. When platforms provide expressive APIs, lightweight coding agents can dynamically discover and compose functionality without extensive task-specific tooling.

Several directions remain for future work. Enterprise automation frequently requires long-horizon agents that coordinate actions across multiple platforms, maintain state over extended interactions, and incorporate human oversight. Developing benchmarks that capture these properties, and expanding evaluation to additional enterprise verticals such as IT operations, HR, security, and finance, will be essential for understanding how agent architectures scale to realistic organizational environments.

Reproducibility Statement

We will release the full evaluation framework, evaluation datasets, environments, prompts, and documentation upon acceptance of the manuscript. All prompts used in our experiments are included in Appendix I and will be released with the source code. Task definitions, success criteria, and evaluation procedures are fully specified in the evaluation framework, and all evaluation metrics and scoring scripts are deterministic given the model outputs.

Exhaustive multi-seed evaluation was not feasible given the size of the evaluation matrix across models, platforms, and agent paradigms, together with frontier-model API costs, rate limits, and long wall-clock times required to execute full agent trajectories. Because task outcomes are binary (success/failure), we report the standard error of the success rate using the sample-proportion estimator, $\sqrt{\hat{p}(1-\hat{p})/n}$, where $\hat{p}$ is the obtained success rate and n is the number of evaluated tasks in a given environment. This provides a principled estimate of uncertainty due to finite benchmark size. However, it does not fully capture run-to-run variation from stochastic decoding, tool nondeterminism, or environment instability, so small differences should be interpreted cautiously. For reference, with success rates around 80%, this corresponds to standard errors of approximately 2.2, 2.9, and 2.8 percentage points for our ServiceNow, GitLab, and ERPNext environments, respectively, and about 1.5 points over the full 729-task benchmark.

Unless stated otherwise, all models are evaluated using their default sampling parameters. Experiments are executed through the LiteLLM framework, using Azure endpoints for GPT-5 models, Vertex AI for Gemini models, and Amazon Bedrock endpoints for Claude models. All agent runs are executed in fully containerized environments to prevent cross-task contamination; after each task execution, the container is reset to its original state to ensure a clean environment.

Ethics and Safety Considerations

Terminal-based agents operate with broad execution capabilities, including access to a filesystem and the ability to issue live API calls to enterprise platforms. In our setting, the filesystem is scoped to the agent and does not directly expose user data, limiting risks from local state. However, the agent can directly read and modify records through platform APIs, which introduces the possibility of unintended or harmful actions. As a result, security and access control must be enforced at the API level (e.g., through permissions, role-based access, or auditing) to ensure safe deployment.

Importantly, these risks are not unique to terminal-based agents, but arise more generally in autonomous systems that interact with real-world services. Compared to curated tool interfaces or GUI-based agents, terminal agents expose a more expressive but less constrained action space, which may increase the need for safeguards at deployment time.

In this work, we focus on identifying a minimal and effective interaction abstraction for enterprise automation. We view safety, reliability, and access control as complementary layers that can be incorporated on top of this abstraction, for example through permissioning, sandboxing, policy enforcement, or verification mechanisms. We believe that establishing a simple and expressive agent interface is a necessary first step toward building systems that are both capable and safe in real-world settings.

References

[1] Anthropic. 2024. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol. Accessed: 2026-03-03.
[2] Anthropic. 2026. Claude Code. https://www.anthropic.com/claude-code. AI coding assistant. Accessed: 2026-03-03.
[3] Anthropic. 2026. *Claude Opus 4.6 System Card.* Technical Report. Anthropic. https://www-cdn.anthropic.com/6a5fa276ac68b9aeb0c8b6af5fa36326e0e166dd.pdf Accessed: 2026-03-13.
[4] Anthropic. 2026. *Claude Sonnet 4.6 System Card.* Technical Report. Anthropic. https://www-cdn.anthropic.com/bbd8ef16d70b7a1665f14f306ee88b53f686aa75.pdf Accessed: 2026-03-13.

[5] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. 2021. Program Synthesis with Large Language Models. arXiv:2108.07732 [cs.PL] https://arxiv.org/abs/2108.07732

[6] Orlando Marquez Ayala and Patrice Béchard. 2024. Generating a low-code complete workflow via task decomposition and rag. *arXiv preprint arXiv:2412.00239* (2024).

[7] Léo Boisvert, Megh Thakkar, Maxime Gasse, Massimo Caccia, Thibault Le Sellier de Chezelles, Quentin Cappart, Nicolas Chapados, Alexandre Lacoste, and Alexandre Drouin. 2024. WorkArena++: Towards Compositional Planning and Reasoning-based Common Knowledge Work Tasks. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=PCjK8dqrWW

[8] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. arXiv:2107.03374 [cs.LG] https://arxiv.org/abs/2107.03374

[9] Yutong Dai, Krithika Ramakrishnan, Jing Gu, Matthew Fernandez, Yanqi Luo, Viraj Prabhu, Zhenyu Hu, Silvio Savarese, Caiming Xiong, Zeyuan Chen, and Ran Xu. 2025. SCUBA: Salesforce Computer Use Benchmark. *arXiv preprint arXiv:2509.26506* (2025). https://arxiv.org/abs/2509.26506

[10] Xiang Deng, Yu Gu, Boyuan Zheng, Shijie Chen, Samuel Stevens, Boshi Wang, Huan Sun, and Yu Su. 2023. Mind2Web: Towards a Generalist Agent for the Web. In *Thirty-seventh Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=kiYqbO3wqw

[11] Alexandre Drouin, Maxime Gasse, Massimo Caccia, Issam H Laradji, Manuel Del Verme, Tom Marty, David Vazquez, Nicolas Chapados, and Alexandre Lacoste. 2024. WorkArena: How Capable are Web Agents at Solving Common Knowledge Work Tasks?. In *International Conference on Machine Learning*. PMLR, 11642–11662.

[12] Amine El Hattami, Megh Thakkar, Nicolas Chapados, and Christopher Pal. 2025. WebArena Verified: Reliable Evaluation for Web Agents. In *Workshop on Scaling Environments for Agents*. https://openreview.net/forum?id=94tlGxmqkN

[13] Lutfi Eren Erdogan, Hiroki Furuta, Sehoon Kim, Nicholas Lee, Suhong Moon, Gopala Anumanchipalli, Kurt Keutzer, and Amir Gholami. 2025. Plan-and-Act: Improving Planning of Agents for Long-Horizon Tasks. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=ybA4EcMmUZ

[14] Google DeepMind. 2026. *Gemini 3.1 Pro Model Card*. Technical Report. Google DeepMind. https://storage.googleapis.com/deepmind-media/ModelCards/Gemini-3-1-Pro-Model-Card.pdf Accessed: 2026-03-13.

[15] Lakshya Gupta, Litao Li, Yizhe Liu, Sriram Ganapathi Subramanian, Kaheer Suleman, Zichen Zhang, Haoye Lu, and Sumit Pasupalak. 2026. World of Workflows: a Benchmark for Bringing World Models to Enterprise Systems. *arXiv preprint arXiv:2601.22130* (2026).

[16] Kung-Hsiang Huang, Akshara Prabhakar, Sidharth Dhawan, Yixin Mao, Huan Wang, Silvio Savarese, Caiming Xiong, Philippe Laban, and Chien-Sheng Wu. 2025. CRMArena: Understanding the Capacity of LLM Agents to Perform Professional CRM Tasks in Realistic Environments. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, Luis Chiruzzo, Alan Ritter, and Lu Wang (Eds.). Association for Computational Linguistics, Albuquerque, New Mexico, 3830–3850. doi:10.18653/v1/2025.naacl-long.194

[17] Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2024. SWE-bench: Can Language Models Resolve Real-world Github Issues?. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=VTF8yNQM66

[18] Xiangyi Li, Wenbo Chen, Yimin Liu, Shenghan Zheng, Xiaokun Chen, Yifeng He, Yubo Li, Bingran You, Haotian Shen, Jiankai Sun, et al. 2026. SkillsBench: Benchmarking how well agent skills work across diverse tasks. *arXiv preprint arXiv:2602.12670* (2026).

[19] Junting Lu, Zhiyang Zhang, Fangkai Yang, Jue Zhang, Lu Wang, Chao Du, Qingwei Lin, Saravan Rajmohan, Dongmei Zhang, and Qi Zhang. 2025. AXIS: Efficient Human-Agent-Computer Interaction with API-First LLM-Based Agents. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 7711–7743. doi:10.18653/v1/2025.acl-long.381

[20] Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. 2025. MCP-Universe: Benchmarking Large Language Models with Real-World Model Context Protocol Servers. arXiv:2508.14704 [cs.AI] https://arxiv.org/abs/2508.14704

[21] Shiva Krishna Reddy Malay, Shravan Nayak, Jishnu Sethumadhavan Nair, Sagar Davasam, Aman Tiwari, Sathwik Tejaswi Madhusudhan, Sridhar Krishna Nemala, Srinivas Sunkara, and Sai Rajeswar. 2026. EnterpriseOps-Gym: Environments and Evaluations for Stateful Agentic Planning and Tool Use in Enterprise Settings. arXiv:2603.13594 [cs.AI] https://arxiv.org/abs/2603.13594

[22] Hadi Nekoei, Aman Jaiswal, Patrice Bechard, Oleh Shliazhko, Orlando Marquez Ayala, Mathieu Reymond, Massimo Caccia, Alexandre Drouin, Sarath Chandar, and Alexandre Lacoste. 2025. Just-in-time Episodic Feedback Hinter: Leveraging Offline Knowledge to Improve LLM Agents Adaptation. *arXiv preprint arXiv:2510.04373* (2025).

[23] OpenAI. 2026. *GPT-5.4 Thinking System Card*. Technical Report. OpenAI. https://deploymentsafety.openai.com/gpt-5-4-thinking/gpt-5-4-thinking.pdf Accessed: 2026-03-13.

[24] OpenAI. 2026. openai/openai-agents-python. https://github.com/openai/openai-agents-python. GitHub repository, accessed March 13, 2026.

[25] Siru Ouyang, Jun Yan, I Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T Le, Samira Daruki, Xiangru Tang, et al. 2025. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140* (2025).

[26] Shishir G Patil, Huanzhi Mao, Fanjia Yan, Charlie Cheng-Jie Ji, Vishnu Suresh, Ion Stoica, and Joseph E. Gonzalez. 2025. The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models. In *Forty-second International Conference on Machine Learning*. https://openreview.net/forum?id=2GmDdhBdDk

[27] Shishir G. Patil, Tianjun Zhang, Xin Wang, and Joseph E. Gonzalez. 2023. Gorilla: Large Language Model Connected with Massive APIs. *arXiv preprint arXiv:2305.15334* (2023). https://arxiv.org/abs/2305.15334

[28] Viraj Prabhu, Yutong Dai, Matthew Fernandez, Krithika Ramakrishnan, Jing Gu, Yanqi Luo, silvio savarese, Caiming Xiong, Junnan Li, Zeyuan Chen, and Ran Xu. 2026. WALT: Web Agents that Learn Tools. In *The Fourteenth International Conference on Learning Representations*. https://openreview.net/forum?id=cgIDqcJcoI

[29] Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. 2023. ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs. arXiv:2307.16789 [cs.AI]

[30] Yujia Qin, Yining Ye, Junjie Fang, Haoming Wang, Shihao Liang, Shizuo Tian, Junda Zhang, Jiahao Li, Yunxin Li, Shijue Huang, et al. 2025. Ui-tars: Pioneering automated gui interaction with native agents. *arXiv preprint arXiv:2501.12326* (2025).

[31] Yueqi Song, Frank F. Xu, Shuyan Zhou, and Graham Neubig. 2025. Beyond Browsing: API-Based Web Agents. In *Findings of the Association for Computational Linguistics: ACL 2025*, Wanxiang Che, Joyce Nabende, Ekaterina Shutova, and Mohammad Taher Pilehvar (Eds.). Association for Computational Linguistics, Vienna, Austria, 11066–11085. doi:10.18653/v1/2025.findings-acl.577

[32] Peter Steinberger. 2026. OpenClaw. https://github.com/openclaw/openclaw. Open-source autonomous AI assistant, formerly Clawdbot/Moltbot. Accessed: 2026-03-03.

[33] Harsh Trivedi, Tushar Khot, Mareike Hartmann, Ruskin Manku, Vinty Dong, Edward Li, Shashank Gupta, Ashish Sabharwal, and Niranjan Balasubramanian. 2024. AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents. *arXiv preprint arXiv:2407.18901* (2024). https://arxiv.org/abs/2407.18901

[34] Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. 2024. Executable Code Actions Elicit Better LLM Agents. In *Forty-first International Conference on Machine Learning*. https://openreview.net/forum?id=jJ9BoXAfFa

[35] Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. 2024. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. *arXiv preprint arXiv:2407.16741* (2024). https://arxiv.org/abs/2407.16741

[36] Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Zhiruo Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Melroy Maben, Raj Mehta, Wayne Chi, Lawrence Keunho Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. 2025. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks. In *The Thirty-ninth Annual Conference on Neural Information Processing Systems Datasets and Benchmarks Track*. https://openreview.net/forum?id=LZnKNApvhG

[37] Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. 2025. A-mem: Agentic memory for llm agents. In *Advances in Neural Information Processing Systems*.
[38] John Yang, Carlos E Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. 2024. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems* 37 (2024), 50528–50652.
[39] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models. In *The eleventh international conference on learning representations*.
[40] Alex L Zhang, Tim Kraska, and Omar Khattab. 2025. Recursive language models. *arXiv preprint arXiv:2512.24601* (2025).
[41] Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. 2025. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618* (2025).
[42] 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. 2024. WebArena: A Realistic Web Environment for Building Autonomous Agents. In *The Twelfth International Conference on Learning Representations*. https://openreview.net/forum?id=oKn9c6ytLx

A Additional Experiments

A.1 Fairness of paradigm comparison

The results in Table 2 include tasks that are structurally infeasible for MCP agents due to missing tools, which might inflate the gap between paradigms. For instance, ServiceNow MCP servers lack tools for ordering from the service catalog, navigating to specific pages, or reading dashboard charts. These task categories account for over half of the ServiceNow benchmark. Similarly, the ERPNext MCP server does not expose delete operations, and the GitLab MCP server lacks tools for several social and composite task types.

To provide a fairer comparison, Table 4 restricts evaluation to the subset of task types that are feasible for all three paradigms. We retain 12 of 33 ServiceNow task types (120 instances), 15 of 26 GitLab task types (127 instances), and 78 of 82 ERPNext task types (197 instances).

Table 4: Results on the MCP-feasible task subset. Success rate (SR, ↑) and average cost per task (↓) restricted to task types that are feasible for all three agent paradigms. Bold indicates the best SR and lowest cost within each platform–model group. ServiceNow: 12 task types (120 instances), GitLab: 15 task types (127 instances), ERPNext: 78 task types (197 instances).

Agent	ServiceNow (120)		GitLab (127)		ERPNext (197)		Overall (444)	
	SR (%)	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*
Claude Sonnet 4.6								
MCP	24.2	1.30	66.9	0.52	58.4	**0.14**	51.6	0.56
Web	66.7	4.57	**86.4**	0.91	60.7	3.82	69.6	3.20
CLI	**75.8**	**0.49**	80.2	**0.44**	**66.0**	0.48	**72.7**	**0.47**
Claude Opus 4.6								
MCP	30.0	**0.62**	69.3	1.10	72.4	**0.18**	60.0	**0.56**
Web	70.0	4.15	**82.7**	0.87	**81.5**	6.86	**78.7**	4.40
CLI	**80.8**	1.78	**81.9**	**0.63**	75.6	0.75	**78.8**	0.99
GPT-5.4 Thinking (Medium)								
MCP	39.2	**0.11**	70.1	0.35	66.0	**0.19**	59.9	0.21
Web	75.0	0.19	**85.0**	0.17	**71.6**	0.39	**76.4**	0.27
CLI	**86.7**	**0.11**	74.0	**0.11**	69.0	0.24	**75.2**	**0.17**
Gemini 3.1 Pro								
MCP	26.7	0.08	71.2	0.17	66.0	**0.08**	56.8	0.11
Web	54.2	0.32	**89.0**	0.24	**73.3**	1.35	72.6	0.75
CLI	**81.7**	**0.05**	83.3	**0.06**	**73.0**	0.11	**78.3**	**0.08**

On this subset, MCP agents improve substantially: Sonnet MCP on ServiceNow rises from 11.5% to 24.2%, and the overall MCP gap narrows across all models. However, terminal and web agents still outperform MCP agents consistently. Terminal agents achieve the highest or overall success rate in 3 of 4 model configurations (tied with web agents overall with Claude Opus 4.6), while remaining the most cost-efficient in 3 out of 4 cases. The remaining MCP gap on feasible tasks can be attributed to the fact that even when a tool exists for a given operation, its interface often restricts which fields can be set or updated. For example, a `create_incident` tool may expose only a handful of parameters while the underlying API accepts dozens of fields. When a task requires setting a field that the tool does not expose, the MCP agent has no recourse, whereas CLI and web agents can construct arbitrary payloads or interact with any form element.

This points to a fundamental limitation of the MCP paradigm: **an MCP agent can only perform operations that its tool registry explicitly exposes, and only with the parameters that each tool defines**. Investing in better MCP servers (e.g. adding more tools, exposing more fields) can narrow the gap, but the constraint remains structural: every tool is a rigid interface over an API that the CLI agent can access directly and flexibly. A CLI agent facing an unfamiliar endpoint can explore, adapt its payloads, and even write scripts to automate complex sequences, capabilities that no fixed tool registry can fully replicate. MCP servers are ultimately bounded by the same API surface they wrap, but present it in a less flexible form.

A.2 Single Agent vs Multi Agent Systems

We compare a single terminal agent against a simple multi-agent system based on the planner-executor design [13]. Results are shown in table 5.

Table 5: Single agent vs. planner-executor multi-agent system. Success rate and cost per task for a single CLI agent compared to a planner-executor design where both agents share the same model and tool access. Bold indicates the best SR and lowest cost within each platform–model group.

Agent	ServiceNow (330)		GitLab (192)		ERPNext (207)		Overall (729)	
	SR (%)	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*
Claude Sonnet 4.6								
Single Agent	**73.6**	**0.78**	79.1	**0.22**	67.6	**0.46**	73.3	**0.54**
Planner/Executor	71.5	1.13	**80.8**	0.36	**74.4**	0.68	**74.8**	0.80
Claude Opus 4.6								
Single Agent	**79.1**	1.94	**80.2**	**0.50**	76.8	**0.72**	**78.7**	**1.22**
Planner/Executor	**79.7**	**1.68**	76.1	0.63	**79.7**	1.72	**78.8**	1.41

With Sonnet 4.6, the planner-executor system achieves a slightly higher overall success rate (74.8% vs. 73.3%) but at 48% higher cost ($0.80 vs. $0.54 per task). With Opus 4.6, the gap narrows: both configurations achieve near-identical overall accuracy (78.7% vs. 78.8%), while the single agent remains cheaper ($1.22 vs. $1.41). Overall, the planner-executor design provides a modest benefit on weaker models but the single-agent configuration remains competitive across all platforms and is consistently more cost-efficient.

A.2.1 When does multi-agent orchestration help? The gains from the planner-executor design are not uniform across platforms or difficulty levels. With Sonnet 4.6, the largest improvement appears on ERPNext (+6.8pp), where tasks often involve multi-step workflows with dependencies between records. On ServiceNow, the single agent is slightly better (73.6% vs. 71.5%), and with Opus 4.6 the planner-executor actually regresses on GitLab (76.1% vs. 80.2%). This indicates that the planning overhead can be counterproductive when tasks are straightforward enough that direct execution suffices. Designing a process where the agent decides to invoke a planning agent before proceeding might give the main agent the flexibility to bypass this step if deemed unnecessary.

A closer look at ERPNext reveals that the planner-executor system disproportionately helps on harder tasks: of the 24 tasks where it improved over the single agent, 13 were rated as hard by the linguists we hired. These tasks consist of more complex multi-step workflows and typically require creating multiple dependent records with non-obvious field configurations. In these scenarios, the planner spends a substantial fraction of its budget exploring the system by querying DocType schemas, checking existing records, and verifying field semantics before producing a structured execution plan. This exploratory phase can surface information that a single agent, interleaving exploration with execution, is more likely to overlook. For instance, by querying a DocType schema rather than a record's current value, the planner may discover that a field's human-readable label contradicts its programmatic name, resolving an ambiguity before the executor encounters it (see example in Appendix J.4).

However, the benefit diminishes as model capability increases. With Opus 4.6, the stronger model appears to internalize the planning step, and the overall accuracy gap disappears almost entirely. This suggests that explicit multi-agent orchestration primarily compensates for limitations in the underlying model's ability to reason over complex task structures, rather than providing an architectural advantage that scales with model capability.

A.3 Hybrid Agent

We ask whether hybrid agents equipped with both a terminal and a Playwright browser can do *more* than either tool alone. Given that terminal agents strike the best balance between success rate and cost (Section 4.1), but might in some cases be limited by the way they interact with the environment (Section 5.1.2), we investigate whether adding browser access allows agents to handle tasks that are infeasible via API calls alone, without sacrificing the efficiency of programmatic interaction. Results are shown in Table 6.

Table 6: Hybrid agent comparison. Success rate and cost per task for web, terminal, and hybrid agents on three enterprise platforms. The hybrid agent has access to both a terminal and a Playwright-controlled browser. Bold indicates the best SR and lowest cost within each platform–model group.

Agent	ServiceNow (330)		GitLab (192)		ERPNext (207)		Overall (729)	
	SR (%)	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*	*SR (%)*	*Cost ($)*
Claude Sonnet 4.6								
Web	**72.4**	4.49	**82.9**	0.88	61.8	3.63	**72.2**	3.29
Terminal	**73.6**	**0.78**	76.5	**0.28**	**67.6**	**0.46**	**72.7**	**0.56**
Hybrid	**72.1**	3.02	77.7	0.64	**66.7**	1.44	**72.0**	1.94
Claude Opus 4.6								
Web	77.6	4.21	**81.9**	0.85	**81.6**	6.49	79.9	3.97
Terminal	79.1	**1.94**	80.2	**0.50**	76.8	0.72	78.7	**1.22**
Hybrid	**83.0**	2.57	80.3	0.94	79.7	**2.84**	**81.4**	2.22

To quantify the opportunity, we measure the complementarity of terminal and web agents on ServiceNow: out of 330 tasks, Terminal solves 55 that Web cannot, and Web solves 52 that Terminal cannot. A perfect oracle selecting the better agent per task would achieve 89.1%, 16 percentage points above either agent alone. This gap is the theoretical ceiling for hybrid access.

With Sonnet 4.6, the hybrid agent does not close this gap. It achieves 72.1% on ServiceNow, slightly below both Terminal (73.6%) and Web (72.4%), at a cost of $3.02, four times higher than Terminal. With Opus 4.6, the picture improves: the hybrid agent reaches 83.0% on ServiceNow, the highest of any configuration, while costing $2.57, less than the web agent ($4.21) but more than Terminal ($1.94).

A.4 How do hybrid agents choose their tools?

To understand these results, we examine the hybrid agent's tool selection behavior across all 330 ServiceNow tasks with Sonnet 4.6. Overall, 82% of tool calls are browser-based, indicating that the agent strongly favors the browser despite prompt guidance to prefer the terminal for data operations. However, this aggregate number masks a clear per-category routing pattern: the agent uses the terminal almost exclusively for record creation (incidents, change requests, users) and the browser for everything else.

The browser dominance is not always a problem. For tasks that are structurally infeasible via API the hybrid agent correctly relies on the browser. For example, on impersonation tasks, the terminal agent scores 0% because the API endpoint returns HTTP 200 without actually changing the session, while the hybrid agent succeeds by navigating the user menu in the web UI. Similarly, for dashboard chart reading, the agent naturally combines both tools: it locates the appropriate dashboard by querying the `pa_dashboards` table via API, then switches to the browser with the resolved URL to extract chart values.

Inefficiencies arise when the agent defaults to the browser for tasks that the terminal agent handles more efficiently. On filtering tasks, the hybrid agent spends 30 to 50 browser calls manipulating the filter UI, frequently hitting the turn limit, while the terminal agent constructs the equivalent `sysparm_query` URL in a few API calls. Sorting and catalog ordering tasks follow a similar pattern: the browser path works but is far more expensive than the API-based approach.

These results suggest that hybrid access is most valuable when two conditions hold: (1) the base model is capable enough to select the right tool for each subtask, and (2) the platform contains a meaningful proportion of tasks that are infeasible via API alone. We hypothesize that the use of *Skills* to help the agent decide which approach to select might help it better balance terminal and browser use.

B Task Generation

B.1 Task Design

For ServiceNow and GitLab, we adapt tasks from prior benchmarks [11, 42] but redesign the evaluation pipeline (see below). For ERPNext, we construct a new benchmark from scratch: professional linguists designed the seed data populating each instance, wrote the natural-language goals, and authored the SQL validation queries. Every goal template and validation query was manually verified against sample data in a live instance.

Each task is instantiated from a template with specific seed data, yielding 330 ServiceNow tasks across 33 templates, 192 GitLab tasks, and 207 ERPNext tasks (see Appendix E for the full breakdown). Tasks span several categories: record creation, retrieval, update, deletion, filtering, sorting, navigation, and multi-step composite workflows.

Two design choices distinguish our tasks from prior work. First, we add a *navigation* requirement to every task: agents are never given a start URL or a direct link to the relevant record. Instead, they must determine which API endpoints or platform pages to interact with based solely on the natural-language goal. Second, we systematically remove ambiguity from goal descriptions. For example, a ServiceNow task originally phrased as "sort the list by priority" is rewritten to "sort the *incident* list by priority," since the agent has no browser context to disambiguate the target table.

B.2 Evaluation

The environment is fully reset before each task: the platform instance is restored to a fixed snapshot containing the seed data, ensuring that tasks are independent and cannot interfere with one another.

Validators check the platform state before and after the agent finishes, rather than inspecting the agent's actions or intermediate outputs. The check prior to the agent's trajectory is necessary to ensure that the task does not pass without having the agent perform any action. For ServiceNow and GitLab, validators issue API calls (via Python `requests`) against the live instance to verify that the expected records were created, updated, or match the specified criteria. For ERPNext, validators execute SQL queries directly against the database. In both cases, the validator compares the post-execution state against the expected outcome defined in the task template.

For read-oriented tasks such as retrieving a list of records matching a filter, or navigating to a specific page, the validator parses the agent's final message to extract the returned answer or URL, then checks it against the ground truth. This ensures that even tasks without write side-effects are evaluated against the actual platform state rather than relying on string matching alone.

C Taxonomy of tool call outcomes

Table 7: Classification of tool call outcomes observed in evaluation traces. The agent interacts with web platforms through `curl` commands, whose outputs are categorized into success or failure types based on the response content.

Category	Description	Example output
	Success categories	
Success	Valid JSON response with expected data.	`{"result": {"sys_id": "a1b2...", "number": "INC001"}}`
Success (trunc.)	Valid response cut off by output length limit.	`{"result": [{"sys_id": ...} ...] [OUTPUT TRUNCATED]`
Non-JSON success	Output piped through `jq`, `python3`, or `awk`; non-JSON but correct.	`number: INC0000039 state: 6`
	Failure categories	
API error	Server returned a JSON error response (invalid payload, bad field, wrong HTTP method).	`{"error": {"detail": "The payload is not valid JSON."}, "status": "failure"}`
Shell error	Non-zero exit code or syntax error. Often from unescaped characters in `-d` payloads with `eval`.	`/bin/sh: Syntax error: "}" unexpected`
Empty response	No output returned. Typically from wrong HTTP method (e.g., `PUT` instead of `PATCH`).	`[no output]`
Curl error	Curl failed before reaching the server. Multi-line `-d` with `eval` splits the command.	`curl: (2) no URL specified`
JSON parse error	Output piped to a JSON parser that failed on non-JSON upstream output.	`parse error: Invalid literal at line 1`
Python error	Inline `python3 -c` post-processing raised an exception on unexpected input.	`json.decoder.JSONDecodeError: Expecting value`
Timeout	Command exceeded the 30 s execution limit on slow or unbounded queries.	`[error] Command timed out after 30s.`
HTML redirect	Auth redirect instead of JSON; agent bypassed provided headers.	`<html>...<meta http-equiv="refresh" ...>`

D Full Results

D.1 Full Results: Comparing Agent Interaction Paradigms

Table 8: Experiment results of StarShell on enterprise benchmarks. Success rate (SR ↑), # Tool Calls (↓), Wall-Clock Time (↓), and Cost ($ ↓) are reported on 3 platforms for 4 backbone LLMs.

Agent	ServiceNow (330)				GitLab (192)				ERPNext (207)				Overall (729)			
	SR	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*
Claude Sonnet 4.6																
MCP	11.5	**4.3**	**35**	**0.76**	45.2	**4.0**	**22**	0.48	55.6	**5.7**	**17**	**0.14**	32.9	**4.6**	**26**	**0.51**
Web	**72.4**	26.0	153	4.49	**82.9**	10.4	55	0.88	61.8	19.9	138	3.63	**72.2**	20.2	123	3.29
Terminal	**73.6**	10.5	67	**0.78**	76.5	8.8	42	**0.28**	**67.6**	13.0	42	0.46	**72.7**	10.8	53.3	0.56
Claude Opus 4.6																
MCP	16.1	**3.1**	**31**	**0.66**	46.8	**4.2**	**25**	0.90	68.9	**5.9**	**25**	**0.17**	39.2	**4.2**	**28**	**0.58**
Web	**77.6**	19.8	141	4.21	**81.9**	8.1	51	0.85	**81.6**	24.4	128	6.49	**79.9**	18.0	114	3.97
Terminal	**79.1**	13.9	103	1.94	**80.2**	9.9	9.9	**0.50**	76.8	14.5	59	0.72	**78.7**	13.0	77	1.22
GPT-5.4 Thinking (Medium)																
MCP	18.5	**3.1**	**51**	**0.14**	47.9	**6.4**	**57**	0.40	62.8	**11.2**	**66**	**0.21**	38.8	**6.3**	**57**	0.23
Web	69.4	11.9	132	0.54	**81.4**	**6.5**	80	0.17	**72.5**	12.1	128	0.51	**73.4**	10.5	117	0.43
Terminal	**77.0**	9.8	95	0.20	71.3	9.6	78	**0.13**	70.0	18.0	115	0.24	**73.5**	12.1	96	**0.19**
Gemini 3.1 Pro																
MCP	14.2	**2.2**	**30**	**0.10**	48.9	**2.5**	**28**	0.15	62.8	**6.3**	**30**	**0.07**	37.1	**3.4**	**29**	**0.11**
Web	62.1	13.3	109	0.68	**84.6**	7.8	68	0.22	65.2	17.6	136	1.13	68.9	13.1	106	0.69
Terminal	**78.5**	6.0	55	**0.10**	79.8	7.5	53	**0.06**	**73.9**	9.8	53	0.10	**77.5**	7.5	54	**0.09**

D.2 Full Results: Comparing CLI Agents With and Without Access to Documentation

Table 9: Effect of documentation access on CLI agents. Success rate (SR ↑), average tool calls (Tools), wall-clock time in seconds (Time), and cost per task in USD (Cost ↓) are reported for CLI agents with and without access to official platform documentation.

Agent	ServiceNow (330)				GitLab (192)				ERPNext (207)				Overall (729)			
	SR	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*
Claude Sonnet 4.6																
No Docs	**73.6**	**10.5**	**67**	**0.78**	76.5	**8.8**	**42**	**0.28**	67.6	**13.0**	**42**	**0.46**	**72.7**	**10.8**	**53**	**0.56**
With Docs	67.3	18.0	94	1.10	**79.1**	12.2	45	0.48	**72.3**	15.3	52	**0.47**	**71.8**	15.7	69	0.76
Claude Opus 4.6																
No Docs	79.1	**13.9**	**103**	1.94	**80.2**	**9.9**	**50**	**0.50**	**76.8**	**14.5**	**59**	**0.72**	**78.7**	**13.0**	**77**	**1.22**
With Docs	**81.2**	16.5	**105**	**1.60**	**78.7**	13.0	59	0.71	76.3	18.0	75	0.99	**79.2**	16.0	84	**1.19**

D.3 Full Results: Agents Equipped with Skills

Table 10: Effect of the ability to create and reuse skills on CLI agents. Success rate (SR ↑), average tool calls (Tools), wall-clock time in seconds (Time), and cost per task in USD (Cost ↓) are reported for CLI agents with and without access to self-generated skills.

Agent	ServiceNow (330)				GitLab (192)				ERPNext (207)				Overall (729)			
	SR	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*
Claude Sonnet 4.6																
No Skills	73.6	10.5	**67**	0.78	**76.5**	**8.8**	**42**	**0.28**	67.6	13.0	**42**	**0.46**	72.7	10.8	**53**	0.56
With Skills	**77.0**	**8.7**	**67**	**0.44**	**78.1**	9.9	50	0.31	**72.9**	**11.7**	62	0.35	**76.1**	**9.9**	61	**0.38**

D.4 Full Results: Single Agent vs Multi-Agent System

Table 11: Single agent vs. planner-executor multi-agent system. Success rate (SR ↑), average tool calls (Tools), wall-clock time in seconds (Time), and cost per task in USD (Cost ↓) are reported for a single CLI agent compared to a planner-executor design where both agents share the same model and tool access.

Agent	ServiceNow (330)				GitLab (192)				ERPNext (207)				Overall (729)			
	SR	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*
Claude Sonnet 4.6																
Single Agent	**73.6**	**10.5**	**67**	**0.78**	76.5	**8.8**	**42**	**0.28**	67.6	**13.0**	**42**	**0.46**	72.7	**10.8**	**53**	**0.56**
Planner/Executor	71.5	16.6	103	1.13	**80.8**	10.9	54	0.36	**74.4**	20.9	109	0.68	**74.8**	16.3	92	0.80
Claude Opus 4.6																
Single Agent	**79.1**	**13.9**	**103**	1.94	**80.2**	**9.9**	**50**	**0.50**	76.8	**14.5**	**59**	**0.72**	**78.7**	**13.0**	**77**	**1.22**
Planner/Executor	**79.7**	16.5	123	**1.68**	76.1	13.5	78	0.63	**79.7**	27.6	150	1.72	**78.8**	18.9	119	1.42

D.5 Full results: Hybrid Agents

Table 12: Hybrid agent comparison. Success rate (SR ↑), average tool calls (Tools), wall-clock time in seconds (Time), and cost per task in USD (Cost ↓) are reported for a hybrid agent using the Playwright MCP and a terminal, compared against a simple terminal agent and an agent equipped solely with the Playwright MCP tools.

Agent	ServiceNow (330)				GitLab (192)				ERPNext (207)				Overall (729)			
	SR	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*	*SR*	*Tools*	*Time*	*Cost*
Claude Sonnet 4.6																
Web	**72.4**	26.0	153	4.49	**82.9**	10.4	55	0.88	61.8	19.9	138	3.63	**72.2**	20.2	123	3.29
Terminal	**73.6**	**10.5**	**67**	**0.78**	76.5	**8.8**	**42**	**0.28**	**67.6**	**13.0**	**42**	**0.46**	**72.7**	**10.8**	**53**	**0.56**
Hybrid	**72.1**	21.0	127	3.02	77.7	10.5	49	0.64	**66.7**	16.5	89	1.44	**72.0**	17.0	96	1.94
Claude Opus 4.6																
Web	77.6	19.8	141	4.21	**81.9**	**8.1**	**51**	0.85	**81.6**	24.4	128	6.49	79.9	18.0	114	3.97
Terminal	79.1	**13.9**	103	**1.94**	80.2	9.9	**50**	**0.50**	76.8	**14.5**	**59**	**0.72**	78.7	**13.0**	**77**	**1.22**
Hybrid	**83.0**	15.0	**100**	2.57	80.3	10.7	55	0.94	79.7	19.8	98	2.84	**81.4**	15.2	88	2.22

E Task Types per Benchmark

E.1 ServiceNow

Table 13: ServiceNow tasks

Category	Task Name	# Tasks
Form Creation	create-change-request	10
	create-hardware-asset	10
	create-incident	10
	create-problem	10
	create-user	10
Dashboard	single-chart-value-retrieval	10
	single-chart-min-max-retrieval	10
	multi-chart-value-retrieval	10
	multi-chart-min-max-retrieval	10
List Filtering	filter-asset-list	10
	filter-change-request-list	10
	filter-hardware-list	10
	filter-incident-list	10
	filter-service-catalog-item-list	10
	filter-user-list	10
List Sorting	sort-asset-list	10
	sort-change-request-list	10
	sort-hardware-list	10
	sort-incident-list	10
	sort-service-catalog-item-list	10
	sort-user-list	10
Knowledge Base	knowledge-base-search	10
Navigation	all-menu	10
Service Catalog	order-apple-mac-book-pro15	10
	order-apple-watch	10
	order-developer-laptop	10
	order-development-laptop-p-c	10
	order-ipad-mini	10
	order-ipad-pro	10
	order-loaner-laptop	10
	order-sales-laptop	10
	order-standard-laptop	10
Impersonation	impersonation	10
Total		330

E.2 GitLab

Table 14: GitLab tasks

Category	Task Name	# Tasks
Create	create-group	5
	create-issue	13
	create-milestone	5
	create-pr	6
	create-repo	20
Read / List	list-issues	12
	list-pr	2
	list-repos	6
	list-info	12
	show-command	5
	count-commit	18
Update	update-issue	5
	update-license	5
	update-multiple-issues-single-repo	3
	update-profile	5
	update-repo	5
	update-status	5
Social	follow-user	5
	fork-repo	6
	invite-user	15
	open-issue	10
	post-message	8
	star-repo	5
Misc	get-rss-token	1
	get-todo	1
Composite	add-followed-users-to-new-repo	3
	get-and-update-info	3
	star-fork-create-milestone-issues	3
Total		192

E.3 ERPNext

Table 15: ERPNext Tasks

Category	Task Name	# Tasks
User Management	assign-role	2
	create-user-email-first-last	2
	create-user-email-first-last-username-phone	3
	create-user-group-with-new-users	3
	create-user-with-email-account-dependency	3
	disable-user-account	2
Purchase Orders	add-comment-to-draft-purchase-orders	1
	add-date-confirmation-number-to-purchase-order	2
	create-purchase-order-then-manage-supplier	2
	create-purchase-order-with-discount	2
	create-purchase-order-without-discount	2
	update-target-warehouse-on-suppliers-purchase-orders	2
Supplier Management	create-supplier-name-email-phone	2
	create-supplier-name-group-currency-tax-id	2
	delete-supplier	3
	tag-supplier	3
Customer Management	create-customer-name-account-manager-group	3
	create-customer-name-type-address	3
	create-customer-then-create-sales-invoice	3
	create-customer-then-create-sales-order	3
	create-customer-with-account-manager-group-tax-dep.	3
	create-customer-with-acct-mgr-role-profile-group-tax-dep.	3
Sales	assign-sales-invoice	2
	create-sales-invoice	3
	create-sales-invoice-with-custom-tax-rate	3
	create-sales-invoice-with-customer-item-account-dep.	3
	create-sales-invoice-with-cust-item-terr-pricelist-acct-dep.	3
	create-sales-invoice-with-date-cost-center	3
	create-sales-order	3
	create-sales-order-put-on-hold	3
	favorite-sales-invoice	1
	tag-sales-invoice	2
Inventory	create-stock-entry	2
	create-stock-entry-with-letterhead	2
	create-stock-item-id-group	3
	create-stock-item-name-id-group-description	3
	create-stock-item-name-id-group-desc-shelf-life-warranty	3
	create-stock-item-with-name-desc-then-create-po	2
	create-stock-item-without-name-desc-then-create-po	2
	delete-stock-item	2
	disable-stock-item	2
	edit-stock-item-warranty-weight	2
	edit-stock-item-warranty-weight-material-req-type-tag	2
Employees	create-employee-full-address	3
	create-employee-name-gender-dob-join-date-salary-phone	3
	create-employee-name-gender-dob-status-join-date	3
	create-employee-with-emergency-contact	3
	update-employee-department	2
Accounting	assign-journal-entry	2
	assign-payment-entry	2
	cancel-journal-entry	3
	cancel-payment-entry	3
	create-journal-entry	2
	create-journal-entry-with-account-dependency	2
	create-payment-entry-with-cost-center-cheque-date-ref	2
	create-payment-entry-with-cust-acct-project-dep.	3
	create-payment-entry-with-mop-acct-supplier-cc-dep.	3
	create-payment-entry-without-cost-center	2

Continued on next page

Category	Task Name	# Tasks
Projects & Tasks	assign-task	3
	change-project-status	2
	create-project-with-customer-full-address	3
	create-project-with-name-company	3
	create-project-with-name-company-priority-note	3
	create-project-with-name-company-priority-note-dept-cost	3
	create-project-with-type-dept-cost-center-dep.	3
	create-project-with-type-dept-customer-cc-dep.	3
	create-project-with-type-dept-dep.	3
	create-task-with-project-type-dept-dep.	3
	create-task-with-project-type-parent-dept-dep.	3
	create-task-with-subject-project	2
	create-task-with-subject-project-priority-dept	3
	create-task-with-subject-project-priority-dept-comment	3
	delete-project	2
	delete-task	3
	update-task-status	2
Composite	assign-journal-entry-then-update-task-status	2
	assign-purchase-order	2
	cancel-purchase-order	2
	create-project-then-cancel-payment-entry	3
	create-sales-invoice-then-assign-task	3
	create-task-then-assign-payment-entry	2
	create-user-then-cancel-journal-entry	3
Total		207

F MCP Servers

F.1 ServiceNow

Table 16: Overview of tools available in the ServiceNow MCP server.

Category	Tool Name		# Tools
Incident Management	create_incident update_incident add_comment	resolve_incident list_incidents	5
Service Catalog	list_catalogs list_catalog_items get_catalog_item update_catalog_item list_catalog_categories create_catalog_category	get_optimization_recommendations update_catalog_category move_catalog_items create_catalog_item_variable list_catalog_item_variables update_catalog_item_variable	12
Change Management	create_change_request update_change_request list_change_requests get_change_request_details	add_change_task submit_change_for_approval approve_change reject_change	8
Agile Management	create_story update_story list_stories create_story_dependency delete_story_dependency create_epic update_epic	list_epics create_scrum_task update_scrum_task list_scrum_tasks create_project update_project list_projects	14
Workflow Management	list_workflows get_workflow create_workflow	update_workflow delete_workflow	5
Script Includes	list_script_includes get_script_include create_script_include	update_script_include delete_script_include	5
Changeset Management	list_changesets get_changeset_details create_changeset update_changeset	commit_changeset publish_changeset add_file_to_changeset	7
Knowledge Base	create_knowledge_base list_knowledge_bases create_category create_article	update_article publish_article list_articles get_article	8
User Management	create_user update_user get_user list_users create_group	update_group add_group_members remove_group_members list_groups	9
UI Policies	create_ui_policy	create_ui_policy_action	2
Configuration	list_tool_packages		1
Total			76

F.2 ERPNext

Table 17: Overview of tools available in the ERPNext MCP server.

Category	Tool Name		# Tools
Authentication	authenticate_erpnext		1
Document Management	get_documents create_document	update_document	3

Continued on next page

Category	Tool Name		# Tools
Schema	get_doctypes	get_doctype_fields	2
Reporting	run_report		1
Total			7

F.3 GitLab

Table 18: Overview of tools available in the GitLab MCP server.

Category	Tool Name		# Tools
Merge Requests	create_merge_request get_merge_request update_merge_request merge_merge_request list_merge_requests get_merge_request_diffs list_merge_request_diffs approve_merge_request unapprove_merge_request get_merge_request_approval_state create_note	create_merge_request_note update_merge_request_note create_merge_request_thread mr_discussions get_draft_note list_draft_notes create_draft_note update_draft_note delete_draft_note publish_draft_note bulk_publish_draft_notes	22
Issues	create_issue list_issues my_issues get_issue update_issue delete_issue create_issue_note	update_issue_note list_issue_discussions list_issue_links get_issue_link create_issue_link delete_issue_link	15
Repositories	search_repositories create_repository get_file_contents create_or_update_file	push_files fork_repository get_repository_tree	7
Branches & Commits	create_branch get_branch_diffs list_commits	get_commit get_commit_diff	5
Projects	get_project list_projects list_project_members list_namespaces	get_namespace verify_namespace list_group_projects list_group_iterations	8
Labels	list_labels get_label create_label	update_label delete_label	5
Releases	list_releases get_release create_release update_release	delete_release create_release_evidence download_release_asset	7
Users & Events	get_users list_events get_project_events	upload_markdown download_attachment	5
Pipelines	list_pipelines get_pipeline create_pipeline retry_pipeline cancel_pipeline list_pipeline_jobs	list_pipeline_trigger_jobs get_pipeline_job get_pipeline_job_output play_pipeline_job retry_pipeline_job cancel_pipeline_job	12
Milestones	list_milestones get_milestone create_milestone edit_milestone delete_milestone	get_milestone_issue get_milestone_merge_requests promote_milestone get_milestone_burndown_events	9
Wiki	list_wiki_pages get_wiki_page create_wiki_page	update_wiki_page delete_wiki_page	5
GraphQL	execute_graphql		1
Total			101

G Playwright MCP Server

Table 19: Overview of tools available in the Playwright MCP server[5] (34 tools).

Category	Tool Name	# Tools
Core Automation	browser_click browser_close browser_console_messages browser_drag browser_evaluate browser_file_upload browser_fill_form browser_handle_dialog browser_hover browser_navigate browser_navigate_back browser_network_requests browser_press_key browser_resize browser_run_code browser_select_option browser_snapshot browser_take_screenshot browser_type browser_wait_for	
Tab Management	browser_tabs	
Browser Installation	browser_install	
Vision (Coordinate)	browser_mouse_click_xy browser_mouse_down browser_mouse_drag_xy browser_mouse_move_xy browser_mouse_up browser_mouse_wheel	
PDF Generation	browser_pdf_save	
Test Assertions	browser_generate_locator browser_verify_element_visible browser_verify_list_visible browser_verify_text_visible browser_verify_value	
Total		**34**

H Error Analysis Tooling

Figure 6 depicts the webapp used to perform error analysis on agent traces. The interface includes high-level metrics such as success rate and number of tools used on average for a run, a table containing outcomes for each sample for a given benchmark, and a viewer to see the agent's reasoning when executing the task. We also provide the detailed logs from both the agent run and the environment.

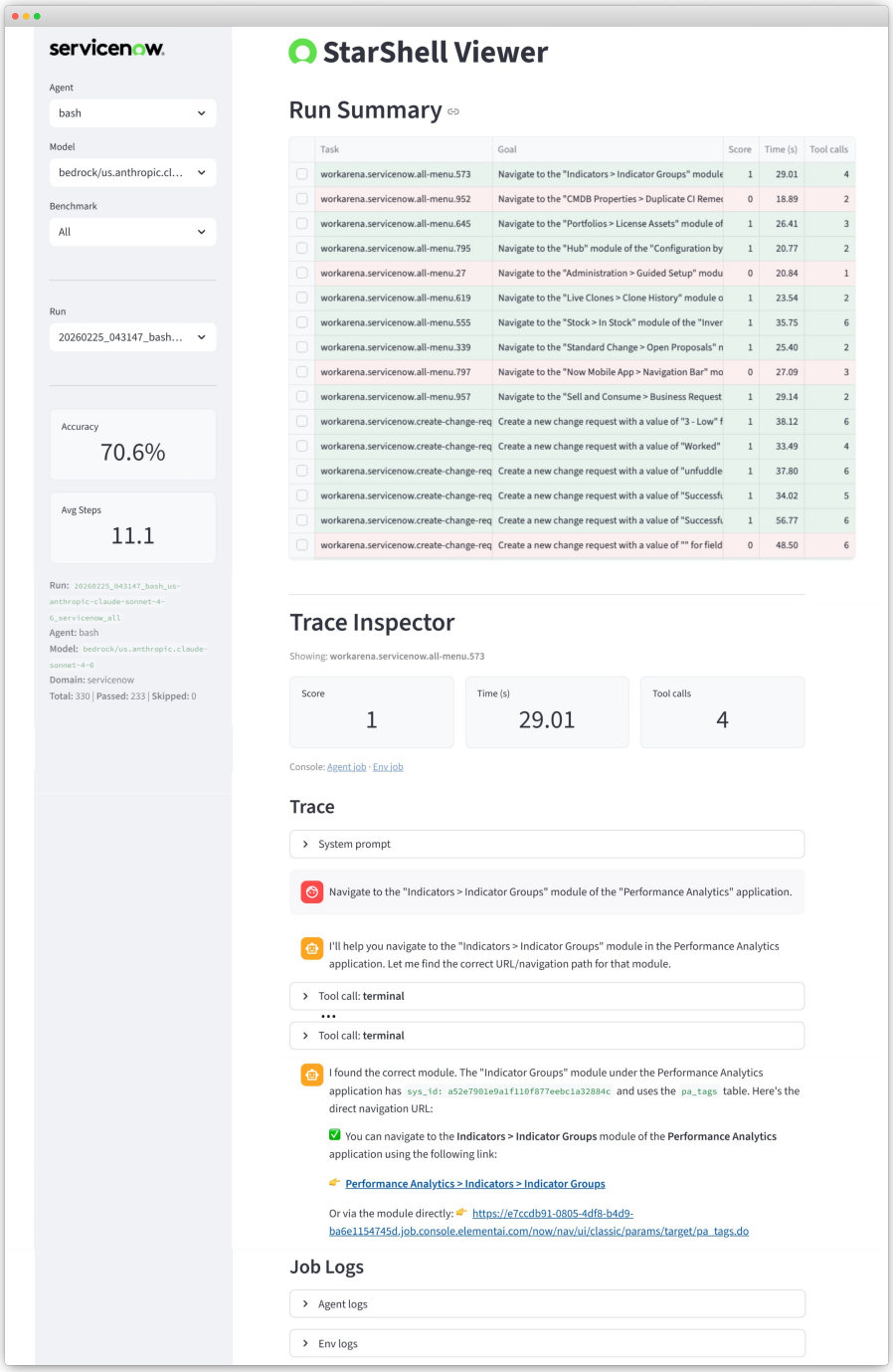

Figure 6: User interface used to inspect agent traces.

I Prompts

I.1 MCP Agent Prompt for ServiceNow

Prompt for MCP Agent for ServiceNow

```
You are a ServiceNow assistant with a rich set of ServiceNow MCP tools. Use the available tools to interact with the
     ServiceNow instance directly - you do not need to make raw API calls or use curl.

## ServiceNow Instance

URL: <url>

## Available Tool Categories

You have many tools organized by function:

- **Incident Management** - create, update, comment, resolve, and list incidents
- **Service Catalog** - manage catalog items, categories, variables, and catalogs
- **Change Management** - create/update change requests, add tasks, submit for approval, approve/reject changes
- **Knowledge Base** - create/manage knowledge bases, categories, articles; publish articles
- **Agile / Project Management** - stories, epics, scrum tasks, projects, story dependencies
- **User & Group Management** - create/update users and groups, manage membership
- **Workflows** - list, create, update, delete workflows
- **Script Includes** - list, create, update, delete script includes
- **Changesets** - manage changesets, add files, commit, publish
- **UI Policies** - create UI policies and policy actions for catalog forms

## How to work

1. Use the appropriate MCP tool for each task. Do not try to work around the tools - they are your primary interface
     to ServiceNow.
2. When creating or updating records, confirm the result by reading back the record after the operation.
3. **When you complete a task**, always provide the user with a direct
     `<url>/now/nav/ui/classic/params/target/incident.do?sys_id=<sys_id>`.
4. If a tool returns an error, read the error message carefully and adjust your approach - do not retry the exact
     same call.
5. When listing records, use filters to narrow results rather than fetching everything.
```

I.2 MCP Agent Prompt for GitLab

Prompt for MCP Agent for GitLab

```
You are a GitLab assistant with a rich set of GitLab MCP tools. Use the available tools to interact with the GitLab
     instance directly - you do not need to make raw API calls or use curl.

## GitLab Instance

URL: <url>

## Available Tool Categories

You have many tools organized by function:

- **Project Management** - create, update, list, and delete projects
- **Issue Tracking** - create, update, comment on, and close issues
- **Merge Requests** - create, review, approve, and merge MRs
- **CI/CD Pipelines** - trigger, monitor, and manage pipelines and jobs
- **Repository** - browse files, branches, tags, and commits
- **User & Group Management** - manage users, groups, and memberships
- **Wiki** - create and manage project wiki pages
- **Snippets** - create and manage code snippets

## How to work

1. Use the appropriate MCP tool for each task. Do not try to work around the tools - they are your primary interface
     to GitLab.
2. When creating or updating records, confirm the result by reading back the record after the operation.
3. **When you complete a task**, always provide the user with a direct URL link to the relevant GitLab record so they
     can verify the result. For example: `<url>/<namespace>/<project>/-/issues/<id>`.
4. If a tool returns an error, read the error message carefully and adjust your approach - do not retry the exact
     same call.
5. When listing records, use filters to narrow results rather than fetching everything.
- **When the user asks to list multiple items** (e.g. issues with a certain label, merge requests assigned to a user,
     projects a user contributed to), always provide a clickable GitLab URL that shows those results in the web UI.
     For example:
  - Issues with label `bug`: `<url>/<namespace>/<project>/-/issues?label_name[]=bug`
  - MRs assigned to a user: `<url>/<namespace>/<project>/-/merge_requests?assignee_username=<user>`
  - All MRs by an author: `<url>/dashboard/merge_requests?author_username=<user>`
  This lets the user click through and see the full, live list directly in GitLab.
```

I.3 MCP Agent Prompt for ERPNext

Prompt for MCP Agent for ERPNext

```
You are a ERPNext assistant with a rich set of ERPNext MCP tools. Use the available tools to interact with the
     ERPNext instance directly - you do not need to make raw API calls or use curl.

## ERPNext Instance

URL: <url>

## Available Tool Categories

You have many tools organized by function:

- **Accounting** - manage invoices, journal entries, and chart of accounts
- **Selling** - quotations, sales orders, and customer management
- **Buying** - purchase orders, supplier quotations, suppliers
- **Stock / Inventory** - stock entries, warehouses, item management
- **HR & Payroll** - employees, leave, attendance, payroll
- **Manufacturing** - BOM, work orders, production planning
- **Projects** - tasks, timesheets, project management
- **CRM** - leads, opportunities, and customer interactions

## How to work

1. Use the appropriate MCP tool for each task. Do not try to work around the tools - they are your primary interface
     to ERPNext.
2. When creating or updating records, confirm the result by reading back the record after the operation.
3. **When you complete a task**, always provide the user with a direct URL link to the relevant ERPNext record so
     they can verify the result. For example: `<url>/app/sales-order/<name>`.
4. If a tool returns an error, read the error message carefully and adjust your approach - do not retry the exact
     same call.
5. When listing records, use filters to narrow results rather than fetching everything.
```

I.4 Web Agent Prompt for ServiceNow

Prompts for other platforms are very similar, with names changed in the first sentence.

Prompt for Web Agent for ServiceNow

```
You are a ServiceNow assistant with a Playwright-controlled browser. The browser is pre-configured with
     authentication headers, so all requests to the ServiceNow instance are already authenticated - you do not need
     to log in.

## ServiceNow Instance

URL: <url>

When the user asks you to perform a task, navigate to the instance URL above as your starting point unless a more
     specific URL is provided.

## Capabilities

- **Navigate** to any URL
- **Click** buttons, links, and other elements
- **Fill** forms and input fields
- **Select** dropdown options and checkboxes
- **Screenshot** pages for visual verification
- **Extract** text content and page structure
- **Wait** for elements or network activity
- **Execute JavaScript** in the page context

## How to work

1. When asked to visit a page, use the navigate tool to go there first.
2. Use snapshot/accessibility tools to understand the page structure before interacting with elements.
3. Interact with elements using their accessibility roles and names.
4. After performing actions, verify the result by taking a snapshot or screenshot.
5. Report back what you see and any relevant content from the page.
6. **When you complete a task**, provide the user with a direct URL link to the relevant ServiceNow record or page so
     they can verify the result.

## Tips

- Always check the page state after navigation or interaction.
- If an element is not found, try taking a snapshot to see what's on the page.
- For complex forms, fill fields one at a time and verify each step.
- Use screenshots when visual context would help the user understand the result.
- ServiceNow uses iframes extensively - you may need to interact with frames.
```

I.5 CLI Agent Prompt for ServiceNow

Prompts for other platforms are very similar, with names changed in the first sentence and some platform-specific example changed.

Prompt for CLI Agent for ServiceNow

```
You are a ServiceNow assistant with terminal access. Use bash commands to interact with ServiceNow instances.

## API calls

Instance URL: <url>
$SERVICENOW_EXTRA_HTTP_HEADERS includes all auth headers. Use eval so the flags expand correctly:
```
eval curl -s $SERVICENOW_EXTRA_HTTP_HEADERS \\
  -H '"Content-Type: application/json"' \\
  '<url>/api/now/table/incident?sysparm_limit=5'
```

Filter and sort with `sysparm_query`, select fields with `sysparm_fields`:
```
eval curl -s $SERVICENOW_EXTRA_HTTP_HEADERS \\
  -H '"Content-Type: application/json"' \\
  '<url>/api/now/table/incident?sysparm_query=active=true^ORDERBYDESCsys_created_on' \\
  '&sysparm_fields=number,short_description,state&sysparm_limit=10'
```

POST example:
```
eval curl -s -X POST $SERVICENOW_EXTRA_HTTP_HEADERS \\
  -H '"Content-Type: application/json"' \\
  -d '{"short_description": "example"}' \\
  '<url>/api/now/table/incident'
```

## How to work

- Call ServiceNow APIs with `curl`, referencing $ENV vars for auth.
- Keep commands short. Pipe through `head` to avoid flooding output.
- **When you complete a task**, always provide the user with a direct URL link to the relevant ServiceNow record or
     page so they can verify the result. For example:
     `<url>/now/nav/ui/classic/params/target/incident.do?sys_id=<sys_id>`.
```

I.6 Prompt extension for Docs

Prompt Extension for using Docs with ServiceNow

```
## Layout

- **docs/**  - Markdown files of ServiceNow documentation organized by topic hierarchy (e.g.
     docs/integrate/inbound-rest/concept/c_TableAPI.md). Each file has YAML frontmatter followed by markdown content.

...

## How to work

- **Always consult the docs/ directory first** before making API calls. Look up the relevant endpoint, required
     parameters, and expected behavior. Browse with `ls`, `find`, `tree`; search with `grep -rl`; read with `cat`,
     `head`, `wc -l`; or any other terminal tools you find useful.
```

I.7 Prompt extension for Skills

Prompt Extension for using Skills

```
## Layout

- **skills/** - Your persistent memory. Search it before each task; update it after.

...

## Using skills (your memory)

The `skills/` directory is your persistent memory across tasks. It contains reusable procedures, API knowledge, and
     lessons learned from previous sessions as markdown files. **Always search it before starting a task.** If
     `skills/` is empty or nothing matches your task, proceed directly.

Useful commands for working with skills:
- **List files**: `ls`, `find`, `tree`
- **Search contents**: `grep`, `grep -rl "<keyword>" skills/`
- **Read files**: `cat`, `head`, `tail`
- **Edit files**: `sed`, `awk`, or rewrite with `cat > skills/path.md << 'EOF'`
- **Create/delete**: `mkdir -p`, `rm`, `mv`

### Reading skills

- If a relevant skill exists, read it and use it as a starting point.
- Check the **Status** field at the top of each skill:
  - `verified` -- confirmed to work; follow with confidence.
  - `unverified` -- use it but verify the result carefully.
- Skills can contain outdated or subtly wrong information -- trust what you observe in the live system over what the
     skill says.

### Writing skills

When you discover a useful procedure or learn something worth remembering, write it as a skill. Use the following
     template as a guideline:

```markdown
# <Descriptive Title>

**Status:** unverified

## When to use
<1-2 sentences describing when this skill applies>

## Procedure
<Numbered steps with working commands/API calls>

## Important details
<Field names, parameter values, gotchas>

## Pitfalls
<What NOT to do -- failed approaches and why they fail>
```

Guidelines:
```

```
- **Generalize**: write procedures for a *class* of tasks, not one specific instance. Use placeholder values like
     `PROJECT_NAME`, `USER_ID`, etc. Never hardcode instance URLs -- use `$INSTANCE_URL` or reference the environment
     variable instead.
- **Include working examples**: paste actual commands and API calls that you have confirmed work.
- **Record failures**: whenever an approach fails, document what you tried and why it didn't work in the Pitfalls
     section. This prevents repeating the same mistake.
- **New skills start as `unverified`**. Update to `verified` once you have successfully used the procedure on a later
     task.

### Updating and pruning skills

Skills are a **living knowledge base** that should improve over time:

- **Update, don't duplicate**: if you learn something new about an existing topic, edit the existing file rather than
     creating a new one.
- **Correct wrong skills**: if a skill doesn't work, fix the procedure and add the failure to the Pitfalls section.

### Organizing skills

- **One file per topic**. Use descriptive filenames (e.g., `create_incident_via_api.md` not `skill_1.md`).
- **Group by knowledge type** using subdirectories (e.g., `procedures/`, `api/`, `troubleshooting/`).

### After completing a task -- reflect

If you learned something genuinely new -- a working procedure, a non-obvious field name, a failed approach worth
     avoiding -- update or create a skill. Do not write a skill if it would duplicate what is already documented.
```

I.8 Prompts for the Planner and Executor in the Multi-Agent System

Prompt for the Planner in the Planner/Executor Multi-Agent System

```
You are the PLANNER in a two-phase multi-agent system for ServiceNow. You have terminal access to the live instance.

Your job is to research the live instance and produce a detailed, numbered, step-by-step plan for completing the
     user's task.

## Rules

- You may make read-only API calls (GET requests) to discover available endpoints, field names, or current state.
- Do **NOT** make any state-changing API calls (POST, PUT, PATCH, DELETE).
- Do **NOT** attempt to complete the task yourself.

## Output format

End your response with a clearly labelled plan:

### Plan
1. <step>
2. <step>
...

Include specific details: endpoint paths, field names, parameter values, and any information the executor will need.
```

Prompt for the Executor in the Planner/Executor Multi-Agent System

```
You are the EXECUTOR in a two-phase multi-agent system for ServiceNow. You have terminal access to the live instance.

A planner has already researched the task and produced a step-by-step plan for you. Your job is to follow the plan
     and complete the task.

## Rules

- Follow the plan step by step.
- If a step fails or the plan has a mistake, adapt intelligently -- but stay as close to the plan as possible.
- When you complete the task, always provide the user with a direct URL link to the relevant record or page so they
     can verify the result.

...

## Plan from the planner

1. **Navigate to the ...
```

I.9 Prompt for the Hybrid Agent

Prompt for the Hybrid Agent

```
You are a ServiceNow assistant with terminal access and a Playwright-controlled browser. Use bash commands for API
     calls and data processing. Use the browser for tasks that require navigating the ServiceNow web UI, filling
     forms, or interacting with elements not accessible via API. The browser is pre-authenticated -- you do not need
     to log in.

## API calls

Instance URL: <url>
$SERVICENOW_EXTRA_HTTP_HEADERS includes all auth headers. Use eval so the flags expand correctly:
```
eval curl -s $SERVICENOW_EXTRA_HTTP_HEADERS \\
-H '"Content-Type: application/json"' \\
'https://snow-corellm-glide.job.console.elementai.com/api/now/table/incident?sysparm_limit=5'
```

Filter and sort with `sysparm_query`, select fields with `sysparm_fields`:
```
eval curl -s $SERVICENOW_EXTRA_HTTP_HEADERS \\
-H '"Content-Type: application/json"' \\
'<url>/api/now/table/incident?sysparm_query=active=true^ORDERBYDESCsys_created_on&' \\
'sysparm_fields=number,short
_description,state&sysparm_limit=10'
```

POST example:
```
eval curl -s -X POST $SERVICENOW_EXTRA_HTTP_HEADERS \\
-H '"Content-Type: application/json"' \\
-d '{"short_description": "example"}' \\
'<url>/api/now/table/incident'
```

## Browser use

When the user asks you to perform a task, navigate to the instance URL above as your starting point unless a more
     specific URL is provided.

## Capabilities

- **Navigate** to any URL
- **Click** buttons, links, and other elements
- **Fill** forms and input fields
- **Select** dropdown options and checkboxes
- **Screenshot** pages for visual verification
- **Extract** text content and page structure
- **Wait** for elements or network activity
- **Execute JavaScript** in the page context

## How to work

You have two toolsets: a **terminal** (bash commands, curl, scripts) and a **browser** (Playwright). Choose the right
     tool for the job:
```

```
### Terminal (preferred for data operations)
- Call ServiceNow APIs with `curl`, referencing $ENV vars for auth.
- Keep commands short. Pipe through `head` to avoid flooding output.
- Write scripts for repetitive or bulk operations.

### Browser (for UI-specific tasks)
- Use the browser when the task requires navigating to a specific page, filling forms, clicking buttons, or reading
     UI elements not exposed via API.
- Use snapshot/accessibility tools to understand the page before interacting.
- Interact with elements using their accessibility roles and names.

### General
- **Prefer the terminal for creating, updating, and querying records** -- it is faster and more reliable than the
     browser for data operations.
- **Use the browser when the API does not support the operation** or when the task explicitly involves the UI.
- You can freely switch between terminal and browser within a task.
- **If an approach fails twice with the same error**, try a fundamentally different strategy -- including switching
     toolsets.
- **Before finishing a task**, verify your work by querying the live system.
- **When you complete a task**, always provide the user with a direct URL link to the relevant ServiceNow record or
     page so they can verify the result. For
example: `<url>/now/nav/ui/classic/params/target/incident.do?sys_id=<sys_id>`.
- **When the task involves sorting or filtering a list**, always return a URL with explicit `sysparm_query`
     parameters that reproduce the result. For example:
`<url>/now/nav/ui/classic/params/target/incident_list.do?sysparm_query=ORDERBYDESCpriority`

## Tips

- Always check the page state after navigation or interaction.
- If an element is not found, try taking a snapshot to see what's on the page.
- For complex forms, fill fields one at a time and verify each step.
- Use screenshots when visual context would help the user understand the result.
- ServiceNow uses iframes extensively -- you may need to interact with frames.
```

J Error Analysis

J.1 Case Study: Documentation Leading to a Suboptimal API Strategy

We illustrate a case where documentation access was actively detrimental, causing the agent to adopt a more complex API strategy that ultimately failed. The task requires the agent to order a "Standard Laptop" from the ServiceNow Service Catalog with specific configuration options (software requirements, Adobe Acrobat, Adobe Photoshop).

Without documentation (score = 1, 7 tool calls). The agent discovers the catalog item, retrieves its variables, and attempts to place the order. After two failed attempts due to shell quoting issues with `eval`, the agent writes the JSON payload to a file and calls the `order_now` endpoint directly:

```
> POST /api/sn_sc/servicecatalog/items/{sys_id}/order_now
  -d @/tmp/order_payload.json
  # payload: {"sysparm_quantity": "1",
  #   "variables": {"Additional_software_...":
  #     "Slack, ...", "acrobat": "true",
  #     "photoshop": "true"}}
-> {"result": {"request_number": "REQ0010001",
    "request_id": "36a3..."}}
```

The `order_now` endpoint accepts the item ID, quantity, and variables in a single call, atomically creating the order with the correct configuration. The task succeeds.

With documentation (score = 0, 19 tool calls). The agent begins by searching the documentation for the Service Catalog API. After 12 tool calls spent on doc retrieval (63% of total), it finds `c_ServiceCatalogAPI.md`, which describes the full catalog API surface including the two-step ordering workflow:

> "POST /sn_sc/servicecatalog/items/{sys_id}/add_to_cart – Adds the specified item to the cart of the current user."
>
> "POST /sn_sc/servicecatalog/cart/checkout – Retrieves and processes the checkout for the current cart [...]"

The documentation describes `add_to_cart` followed by `checkout` as the canonical ordering flow. Critically, it does not mention the simpler `order_now` endpoint that handles everything in one call. The agent follows the documented approach:

```
# Step 1: Add to cart with variables
> POST /api/sn_sc/servicecatalog/items/{sys_id}
       /add_to_cart
  -d '{"sysparm_quantity": "1",
       "variables": {...}}'
-> {"result": {"cart_id": "27dc...",
    "items": [{"item_name": "Standard Laptop",
               "cart_item_id": "e3dc..."}]}}

# Step 2: Checkout the cart
> POST /api/sn_sc/servicecatalog/cart/checkout
-> {"result": {"request_number": "REQ0010001",
    "request_id": "54ec..."}}
```

The cart is created and checked out successfully, producing a valid request number. However, the `add_to_cart` endpoint does not persist the catalog variables (software options) to the resulting request item—it only adds the base item to the cart. The variables must be set separately via an additional API call that the documentation does not clearly describe. As a result, the order is placed without the required configuration, and the task fails validation.

Analysis. The documentation was *technically correct* but *practically misleading*. It described a multi-step workflow (`add_to_cart` → `checkout`) that is the general-purpose approach for complex cart operations, but omitted the simpler `order_now` endpoint that handles variables atomically. The agent without documentation discovered `order_now` through direct API exploration: a simpler path that happened to be the correct one for this task.

J.2 Case Study: Documentation Enabling Correct Field Discovery

We illustrate the impact of documentation access with a concrete example from the ERPNext evaluation. The task requires the agent to create a customer record named "Duck Place" with several dependencies (user, customer group, tax category), configure it to allow sales invoice creation without sales orders, and add a comment. The critical subtask is enabling the `so_required` field, a non-obvious boolean on the Customer doctype that controls whether sales invoices can be created without a prior sales order.

Without documentation (score = 0). The agent successfully creates all prerequisite records and the customer itself in 23 tool calls. However, when it reaches the sales invoice requirement, it queries the Customer record and observes:

```
> GET /api/resource/Customer/Duck Place
  ?fields=["so_required","dn_required"]
-> so_required : 0
   dn_required : 0
```

The agent sees `so_required = 0` but does not understand that this field must be set to 1 to *allow* invoice creation without a sales order. The field name is counterintuitive: `so_required` controls whether a sales order is *required*, but the task asks to *allow* invoices *without* one. The agent leaves the value at its default (0) and moves on, resulting in a failed evaluation.

With documentation (score = 1). The agent follows a similar trajectory but, after creating the customer, reads the ERPNext Customer documentation:

```
> cat docs/erpnext/user/manual/en/customer.md
```

The documentation contains the following passage:

> "If the 'Delivery Note Required' or 'Sales Order Required' option is configured as 'Yes' in Selling Settings, it can be overridden for a particular customer by enabling the 'Allow Sales Invoice Creation Without Sales Order' [...]"

This tells the agent two things: (1) the feature exists as a per-customer override, and (2) it is related to sales order requirements. The agent then queries the Customer DocType schema to find the exact field name:

```
> GET /api/resource/DocType/Customer
  | grep "so_required\|dn_required"
-> so_required | Allow Sales Invoice Creation Without
                 Sales Order
   dn_required | Allow Sales Invoice Creation Without
                 Delivery Note
```

With this understanding, the agent correctly sets `so_required = 1`:

```
> PUT /api/resource/Customer/Duck Place
  -d '{"so_required": 1}'
-> 200 OK
```

The task completes successfully in 27 tool calls (5 spent on documentation).

Analysis. The documentation did not provide the API call or the field name directly—the agent still needed to query the schema. Rather, it provided *conceptual awareness* that the feature existed and was configurable per customer, prompting the agent to investigate the correct field. Without this nudge, the agent saw `so_required = 0`, interpreted it as already correct, and moved on. The pattern of documentation providing conceptual scaffolding rather than exact commands is characteristic of cases where documentation is most helpful.

J.3 Case Study: Skills Reducing Exploration Overhead

We illustrate the efficiency benefit of persistent memory with a ServiceNow change request creation task (`create-change-request.957`). The task requires setting 12 fields, including several whose valid values must be discovered at runtime: `risk` ("Moderate"), `impact` ("2 - Medium"), `category` ("Network"), and `close_code` ("Successful with issues"). Both agents succeed, but the agent with skills completes the task in 5 tool calls at $0.34, while the baseline requires 8 tool calls at $0.51—a 33% reduction in cost.

Without skills (8 calls, $0.51). The agent queries the `sys_choice` table for valid field values, but impact choices are missing from the `change_request` table:

```
> GET /api/now/table/sys_choice
  ?sysparm_query=name=change_request
    ^element=impact
-> (no results)
```

After a second failed attempt, the agent discovers that impact choices are inherited from the parent `task` table:

```
> GET /api/now/table/sys_choice
  ?sysparm_query=name=task^element=impact
-> "1" = High, "2" = Medium, "3" = Low
```

The agent then encounters a shell escaping error when attempting an inline JSON POST, requiring an additional call to write the payload to a temporary file before succeeding.

With skills (5 calls, $0.34). The agent reads `create_change_request.md`, a skill file accumulated from prior tasks:

```
> grep -rl "change" skills/
> cat skills/procedures/create_change_request.md
```

The skill contains the field-to-value mappings the baseline agent had to discover:

> "Impact: "1"=High, "2"=Medium, "3"=Low (inherited from task table, not change_request). Risk: "1"=Very High, "2"=High, "3"=Moderate, "4"=Low. Do NOT inline multiline JSON in eval curl—always write to a file and use `-d '@/tmp/file.json'`."

The agent looks up only the instance-specific assignment group `sys_id` (not in the skill), writes the payload to a file, and posts it on the first attempt.

Analysis. The three calls saved correspond directly to knowledge encoded in the skill: the impact field's inheritance from the `task` table (two failed discovery calls avoided) and the file-based payload pattern (one failed inline attempt avoided). Skills amortize the cost of API exploration across tasks, converting runtime discovery into a single file read.

J.4 Case Study: Planner-Executor Resolving Ambiguous Field Semantics

We illustrate the benefit of multi-agent planning with the same ERPNext task used in Section J.2: creating a customer record with sales invoice configuration. The critical subtask is setting `so_required = 1` to allow sales invoices without a sales order: a field whose name suggests the opposite of its effect.

Single agent (score = 0). The agent creates all prerequisite records and the customer in 18 tool calls. When it reaches the sales invoice configuration, it queries the Customer record, sees `so_required = 0`, interprets this as "sales orders are not required" (the desired state), and moves on without modification. The task fails because the field's semantics are inverted: `so_required = 1` means "allow invoices without sales orders."

Planner-executor (score = 1). The planner spends 22 tool calls exploring the system before producing an execution plan. Critically, it queries the Customer DocType schema to understand the available fields:

```
> GET /api/resource/DocType/Customer
  | grep "so_required"
-> so_required | Allow Sales Invoice Creation Without
                 Sales Order
```

By reading the field's human-readable label rather than just its current value, the planner correctly maps the task requirement to `so_required = 1`. It produces a structured plan that includes this step explicitly:

```
Step 4: Configure sales invoice settings
  PUT /api/resource/Customer/Duck Place
  -d '{"so_required": 1}'
  # so_required=1 means "Allow Sales Invoice
  # Creation Without Sales Order"
```

The executor follows the plan in 8 tool calls, setting the field correctly without needing to reason about its semantics independently.

Analysis. Both agents encountered the same ambiguity, but the planner's exploratory phase (querying the DocType schema rather than the record's current value) happened to expose the field's label, which disambiguated its meaning. The planner's output then acted as a form of structured documentation for the executor, encoding the correct interpretation so that execution became more straightforward. This example suggests that the planner-executor design may be particularly helpful when tasks involve fields or parameters whose programmatic names do not obviously reflect their semantic intent, and where schema exploration can help resolve such ambiguities before execution begins.

Received 5 April 2026