WEBMCPWORLD: CAN LLMS SYNTHESIZE WEBMCP TOOLS TO FINISH OPEN-ENDED TASKS IN WEB APPS?

Anonymous authors
Paper under double-blind review

ABSTRACT

Large language models (LLMs) have demonstrated remarkable capability at both writing code and operating computers. In real-world scenarios, users make colloquial requests and let an agent finish computer-use tasks; for a frequently repeated workflow, they prefer to build a reusable tool(e.g., a skill, an MCP server), so the work runs stably instead of being re-derived. Existing works prove that agents given MCP or CLI tools alongside GUI actions succeed more often and in fewer steps. Yet benchmarks have not kept pace: SWE-bench scores a patch to a named repository, application-generation benchmarks assess a model's ability to generate complete applications, and computer-use benchmarks measure GUI agents' success rates on individual computer-use tasks. None of them evaluates this ability—turning a colloquial request into a small, working, reusable tool over an application that already exists.

To address this gap, we introduce **WebMCPWorld**, a benchmark consisting of 50 real-world web applications and 1280 colloquial user queries. Given the source code of a web application and a query, a model must do two things: locate the application functionality that the query targets, and turn that part of its source into the WebMCP tool layer, and declare a parameter interface that a runtime must later bind the query's implicit arguments to, without ever observing that the runtime or its inputs. An execution-based runtime judges the synthesized tooling both when supplied with the query's true parameters and on whether the user's query is ultimately fulfilled. WebMCPWorld remains highly challenging for current frontier models. Across 18 configurations of 10 LLMs, even the best-performing setup, Claude Opus 5 with Claude Code, achieves a mean end-to-end completion rate of only 10.0% across all queries. These results highlight the difficulty of turning colloquial requests into reliable, reusable tools for existing web applications.

1 INTRODUCTION

Large language models are increasingly integrated into daily workflows, and users are likely to have them handle routine tasks like writing code and operating the computer. Reusable tools let agents perform recurring workflows more deterministically. MCP standardizes access to these tools (Anthropic, 2024; Model Context Protocol Contributors, 2025), and recent evaluations report gains of tools in task success and interaction efficiency (Jia et al., 2026; Prabhu et al., 2026). WebMCP (Web Machine Learning Community Group, 2026) extends this approach to web applications, exposing functionality as callable interfaces for tasks such as tracking shipments or checking forecasts. But these interfaces still require implementation to connect application functionality to users' requests.

Models acting as *tool-makers* could automate this implementation (Cai et al., 2023; Qian et al., 2023; Yuan et al., 2024). Given a user's query and application source code, a model could identify the required functionality and compose a WebMCP tool layer: tool definitions and workflow code that invokes them. The evaluation question is whether models can reliably turn ordinary requests into working, reusable tooling over applications that already exist.

Existing benchmarks do not directly test this capability. Code benchmarks either focus on repository repair like SWE-bench (Jimenez et al., 2024), or whole-application generation based on the user

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

request (Ran et al., 2025; Lu et al., 2025; Liu et al., 2026). Computer-use and MCP benchmarks assess interaction through GUIs or pre-existing tools (Zhou et al., 2024; Xie et al., 2024; Kaminski et al., 2026), while tool-creation studies cover functions and APIs (Cai et al., 2023; Wang et al., 2025a). Tool-Genesis focuses on MCP server synthesis from abstract requirements, rather than browser-side tool integration into existing web applications.

To investigate this question, we introduce **WebMCPWorld**, a benchmark of 50 web applications, 258 business features, and 1280 colloquial queries. Figure 1a outlines how repository screening, GUI exploration, and expert review produce these tasks. Each feature corresponds to a GUI action sequence that accomplishes a business operation. Each instance provides a user query, the source code, and a description of a web application. The model selects application features, resolves implicit parameters, and creates the reusable tool to meet the query's needs. The number of query variants differs across features. Tasks are *open-ended in their solution paths*: during sandbox execution, the end-to-end agent chooses workflow calls, arguments, and their order without a prescribed reference trajectory, while the task objective and acceptance criteria remain fixed.

As illustrated in Figure 1b, generated workflows are executed with reference arguments and separately tested by an end-to-end agent. Query variants and application-state resets support evaluation of usability and reuse. Execution traces support failure analysis, while trajectory length and cost characterize efficiency.

Under this protocol, we evaluate ten LLMs, including Claude Opus 5 (Anthropic, 2026b) and DeepSeek V4.1 Flash (DeepSeek, 2026), in 18 configurations using Claude Code (Anthropic, 2025), Codex (OpenAI, 2025), and mini-swe-agent (SWE-agent Team, 2025), and obtain three main findings. First, the best mean end-to-end completion rate is only 10.0% over all queries, versus 4.45% for GUI-only (Table 1); unequal budgets limit causal comparisons. Second, 95.0% of feature–configuration–repetition groups have no successful end-to-end query, showing limited success across query variants (Figure 4). Third, 62.4% of retained agent-stage records contain arguments that heuristic matching cannot verify as correct, making this the largest non-pass category without establishing the cause of failure (Figure 6). Together, these findings motivate evaluating generated tools through completed application tasks and repeated use, beyond artifact production.

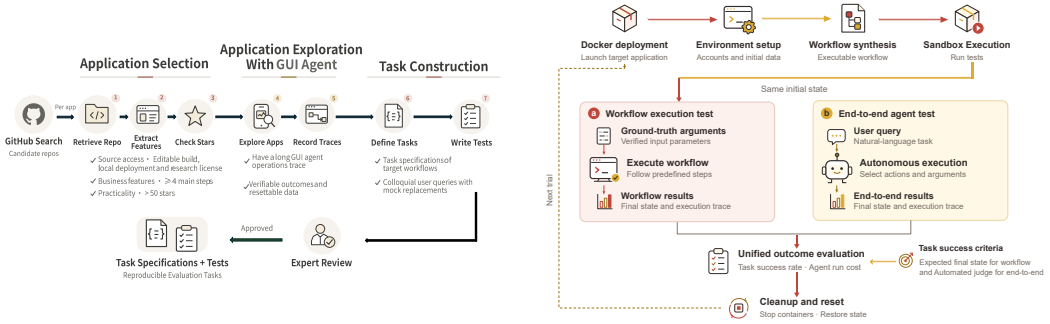
We summarize our main contributions as follows:

- ***New real-world problem:*** We study WebMCP tool synthesis from colloquial requests and application source code.
- ***New Benchmark:*** We introduce WebMCPWorld, covering 50 web applications with separate tests for tool correctness and task completion.
- ***Evaluation Results and Analysis:*** We evaluate 18 combinations of 10 models and 3 agent harnesses, analyzing their limited performance and failure patterns in real-world computer-use scenarios.

2 Related Work

MCP and WebMCP. The Model Context Protocol (MCP) provides a common interface for agents to access external tools and data (Anthropic, 2024; Model Context Protocol Contributors, 2025). Surveys examine its architecture, interoperability, security, and software design (Hou et al., 2025; Ray, 2025; Ehtesham et al., 2025; Sarkar & Sarkar, 2025). WebMCP brings this tool-interface approach into web applications: its proposed browser API exposes application functionality as JavaScript tools with natural-language descriptions and structured input schemas (Web Machine Learning Community Group, 2026). Agents can invoke these operations within the application's existing interface, for example to search records or submit a form.

Tool-Genesis already evaluates reusable MCP tool generation from abstract requirements without preset schemas, including functional correctness and downstream utility (Xia et al., 2026). WebMCPWorld instead grounds tool synthesis in existing web-application source and colloquial requests, then checks task outcomes in the deployed application's state. This is a distinction in setting, not the introduction of tool creation or downstream evaluation. TM-Bench studies scientific-repository tools (Wölflein et al., 2025); other systems convert OpenAPI specifications or repositories into MCP servers (Mastouri et al., 2025; Ouyang et al., 2025).



(a) Task construction. (b) Execution and evaluation.

Figure 1: Overview of WebMCPWorld. (a) Construction of WebMCPWorld. (b) The evaluation pipeline of WebMCPWorld.

Benchmarks for code generation. SWE-bench evaluates code changes that resolve issues in a specified repository (Jimenez et al., 2024). AppForge, WebGen-Bench, and WebCoderBench evaluate application generation from requirements (Ran et al., 2025; Lu et al., 2025; Liu et al., 2026). CodeClash evaluates iterative code development toward high-level objectives (Yang et al., 2025). These benchmarks assess the correctness or utility of generated code. WebMCPWorld focuses on a distinct output: reusable tools that operate an existing web application and support user task completion.

Benchmarks for tool use and MCP. ToolBench, introduced with ToolLLM, and API-Bank evaluate tool selection, planning, and API calls over provided interfaces (Qin et al., 2024; Li et al., 2023). MCP-Bench, MCP-Universe, and MCPMark extend tool-use evaluation to workflows over existing MCP servers (Wang et al., 2025b; Luo et al., 2025; Wu et al., 2025). LiveMCPBench examines selection from large tool inventories (Mo et al., 2025), while MCP-AgentBench and MCPAgentBench assess task outcomes and execution efficiency (Guo et al., 2025; Liu et al., 2025). Their tools are supplied to the evaluated agent. WebMCPWorld additionally requires the evaluated model to synthesize the tooling and tests both its behavior with verified arguments and its use to fulfill a natural-language request.

Effectiveness of tools for agents. Beyond evaluating tool-use competence, studies examine how access to tools affects agent performance. OSWorld-MCP equips computer-use agents with validated tools and reports gains in task success (Jia et al., 2026). WALT discovers reusable tools through live website exploration and demonstrates higher success with fewer interaction steps (Prabhu et al., 2026). These studies establish the value of tool libraries prepared before evaluation. WebMCPWorld complements them by testing whether the evaluated model can create effective tools from application source and user requests.

3 WebMCPWorld

WebMCPWorld evaluates whether a model can turn a colloquial user request into executable WebMCP tooling over existing web applications. The benchmark has four stages: constructing tasks and their tests, synthesizing tools and workflow code, running that code in a sandbox, and judging the resulting outcomes. Figure 1 juxtaposes task construction and evaluation; Figure 2 follows a concrete Chatwoot task through synthesis, execution, and judging.

3.1 Construction

As shown in Figure 1a, construction begins with GitHub search, repository retrieval, feature extraction, and star-count checks. We screen candidates using the checklist in Appendix A, which covers four aspects:

- **Source access and instrumentation.** Complete frontend source must expose pages, components, event handlers, and request logic, and the build pipeline must permit instrumentation or source analysis.
- **Reproducible evaluation.** Applications must support local deployment with pinned versions, resettable data, controllable external dependencies, and outcomes verifiable through the UI, APIs, database, or logs.
- **Business coverage.** Each application must provide at least four distinct business features, including one or two involving cross-page, cross-object, or consecutive state transitions; showcase sites, games, and simple demos are excluded.
- **Research use and community interest.** Licenses must permit local deployment, source analysis, and research experiments, with special restrictions flagged; repositories must have more than 50 GitHub stars at screening time.

Each extracted feature has a name and a description summarizing the *main steps* of its GUI action sequence. For example, `crud_console` describes building a CRUD management console: connect a database or API, create queries, build table and form components, bind create/read/update/delete operations, and deploy the console. These main steps describe a business operation at a higher level than individual clicks or keystrokes; they do not prescribe the end-to-end agent's execution trajectory.

For applications that pass screening, a GUI agent explores these features and records interaction traces with observable outcomes. The traces ground the main steps in operations that can be executed and checked in the application.

The recorded GUI action sequences are then converted into task specifications and tests. Each specification pairs colloquial user queries with a target feature and its expected final state; mock replacements support controlled, repeatable execution. Human reviewers confirm that every feature is accessible through the GUI and review its specification and tests before approval. The benchmark contains 50 applications, 258 business features, and 1,280 queries, with varying query counts per feature. Application profiles and difficulty annotations appear in Appendix B; Figure 7 in Appendix D shows the query distribution across application categories.

3.2 Synthesis

The approved tasks enter the evaluation pipeline in Figure 1b. The target application is first deployed in Docker, and its accounts and initial data are configured. This preparation establishes the application state used for the subsequent tests.

For the target application, the synthesis agent uses its source code and task queries to identify the required functionality and produce executable workflows exposed through WebMCP tools. It defines the tool interface, parameter contract, and code that implements the requested operations. The synthesized artifact is then passed to sandbox execution, where its behavior is evaluated both with verified arguments and through an agent responding to the user query. Prompt templates and synthesis budgets are provided in Appendix C.

Figure 2 follows a Chatwoot contact-and-conversation task; Appendix C.3.1 shows its runtime tool schema and call.

3.3 Sandbox Run

The sandbox evaluates the synthesized artifact in two separate tests with application state reset before each eligible run (Figure 1b). The *workflow execution test* supplies verified ground-truth arguments and executes the workflow's predefined steps. This tests whether the generated workflow implements the intended operation when its inputs are specified.

The *end-to-end agent test* supplies the query and function schemas built from the page's registered workflows: names, descriptions, and typed parameters. The user message also lists these workflows. This registry-to-model bridge and a concrete invocation are detailed in Appendix C.3.1. The agent chooses workflow calls, arguments, and their order based on execution results. This solution path is open-ended within fixed task objectives, acceptance criteria, tools, and budgets. Direct clicking and

User query

```
app             chatwoot
query id        chatwoot.contact.create_and_start_conversation.v1
text            “Create contact “Lin Chen”, email lin.chen+cw1@example.test, phone
                +86-13800001001, and start a conversation in “Website Inbox” whose
                subject is “Order delay”.”
params          name · email · phone · inbox · subject
```

⬇

Workflow synthesis

```
harness · model Mini-SWE-Agent  ·  deepseek-v4.1-flash        iteration 1 of 3
input           the application's source code
produces        workflow_create_contact_and_conversation
declares        name · email · phone · inbox · subject
```

⬇

Workflow execution test

```
params          translated input-by-input from the query:
                name=Lin Chen   email=lin.chen+cw1@example.test   phone=+86-13800001001
                inbox=Website Inbox   subject=Order delay
result          success=true   contact_id=312   conversation_id=1241   dom_verified=true
cost            88 steps · 7.2 s        judge: pass
```

⬇

End-to-end agent test

```
sees            tool schemas for the registered workflows
calls           workflow_create_contact_and_conversation(
                {name, email, phone, inbox, subject})   -- one call, the same five values
ends            success=true   error=""
cost            89 steps · 2 turns        judge: pass
```

⬇

Unified outcome evaluation

```
label           pass        confidence 0.95
evidence        page state yes   screenshot yes        screenshot_agrees = true
reason          “The final page shows conversation #2428 in Website Inbox for contact
                Lin Chen with the email lin.chen+cw1@example.test, phone +8613800001001,
                and the opening message Order delay — exactly what was requested.”
read off image  Lin Chen / #2428 / Website Inbox / Order delay /
                lin.chen+cw1@example.test / +8613800001001
```

Figure 2: A Chatwoot example from user request to judged outcome.

typing are reserved for the GUI-only baseline. Pinned execution requires mapped reference inputs; unmapped inputs may still be tested by the agent. Agent execution is conditional on the runtime eligibility rules, so the two phases cover different subsets. Each executed run records its final state and trace; Section 4.1 specifies the reporting denominators.

3.4 JUDGE

The two sets of execution results enter a unified evaluation stage. For the workflow test, the recorded final state is checked against the expected state specified by the task. For the end-to-end test, an automated judge uses the user query, task success criteria, final state, and execution trace to determine whether the request was fulfilled. Revised judging prioritizes final-state evidence; intermediate errors alone do not force failure. Both modes assess the same task objective, while differing in whether the runtime or the agent supplies the operations' arguments. In the example, the judge checks evidence of the requested contact and conversation, including the contact details, inbox, and subject (Figure 2).

We report task success separately for the two modes, synthesis usage in Appendix E, and GUI execution effort in Appendix B. Category composition and supplementary analyses appear in Appendix D. After outcome evaluation, containers are stopped and application state is restored. The reset starts the next trial, completing the evaluation cycle shown in Figure 1b.

4 EXPERIMENTS

4.1 SETUP

Benchmark and configurations. WebMCPWorld contains 50 applications, 258 features, and 1,280 queries, with varying query counts per feature (Appendix D). We evaluate ten models: Claude Opus 4.8 (Anthropic, 2026a), Opus 5, and Sonnet 5 (Anthropic, 2026c); GPT-5.6 Sol, Terra, and Luna (OpenAI, 2026); Qwen3.8 Max (Alibaba Cloud, 2026); Kimi K3 (Moonshot AI, 2026); GLM-5.3 (Z.ai, 2026); and DeepSeek V4.1 Flash. The 18 configurations use Claude Code (CC), Codex (CX), and mini-swe-agent (MSA). Three repetitions yield 69,120 planned query cells; GUI-only runs once. DeepSeek V4.1 Flash is the fixed runtime agent and judge. The nominal synthesis setting `--max-steps=50` is not enforced by the CLI: Claude and mini use 20-read/30-step checks, while Codex relies on prompts. Budgets and prompts are documented in Appendix C.

Judging and success metrics. All results use revised, outcome-based judge labels: an intermediate execution error does not force failure when the independent final-state evidence establishes task completion. We distinguish *query-to-completion success rate (QCSR)*, our overall metric, from *agent-stage success rate (ASR)*, a conditional metric. For each configuration and repetition, let N be all 1,280 planned queries, A be the number of retained agent-stage trials, and S be the judge-confirmed completions. Then

$$\mathrm{QCSR} = 100\,S/N, \qquad \mathrm{ASR} = 100\,S/A.$$

Both metrics use the same success count. QCSR covers the complete process from query to completion: queries without synthesized tools or retained agent-stage trials remain in its denominator. ASR conditions on entering the agent stage and uses 34,924 retained records overall, excluding skipped trials and 5,177 pre-skip-rule measurements. These excluded measurements contribute no successes to either metric; QCSR keeps the planned-query denominator fixed. Incomplete reference-input mapping does not itself preclude an agent-stage attempt or a judge-confirmed completion. Workflow success is a separate metric: both its numerator and denominator require artifacts and fully mapped inputs (36,226 eligible cells overall). Eligible timeouts and unresponsive pages remain in the relevant denominators. Table 1 reports min/mean/max of per-repetition rates; GUI comparisons use QCSR. Appendix D.2 gives set definitions and distinguishes these means from pooled rates.

4.2 MAIN RESULTS

Completion remains limited. MSA–DeepSeek has the highest mean conditional workflow rate, 8.4% (range 7.6–9.9%); CC–Opus 5 leads ASR at 19.1% (15.8–21.9%). For Qwen3.8 Max, mean ASRs are 2.8%, 8.3%, and 12.3% with CC, CX, and MSA. These configuration differences combine synthesis, budget enforcement, and eligibility effects rather than isolate a harness contribution.

Coverage and GUI comparison. The corrected ledger partitions 69,120 cells into 1,628 without tools, 31,266 without complete input mapping, and 36,226 eligible workflow cells, including 813 judge passes (Table 9). The retained agent stage has 1,644 passes out of 34,924 records (pooled ASR: 4.7%). In QCSR, CC–Opus 5 averages 10.0% (7.4–11.8%), the only configuration mean above GUI-only's 57/1,280 (4.45%). MSA–Qwen and MSA–DeepSeek both average 4.3%. Equal query denominators do not equalize action granularity, synthesis effort, or runtime budgets; GUI also has only one repetition (Appendix B).

Stage coverage and overlap. Figure 3 shows 36,226 workflow-eligible cells and 34,924 measured agent cells, or 52.4% and 50.5% of all planned queries. These sets overlap in 3,658 cells and are not nested: the agent also receives 31,266 cells without complete reference-input mapping, while 32,568 eligible workflow cells do not enter the retained agent evaluation. The agent's 1,644 passes yield a pooled ASR of 4.7% and QCSR of 2.4%. Thus, the two tests measure different populations;

Table 1: Pass rates (%): min/mean/max over three repetitions. Workflow conditions on complete input mapping; ASR conditions on agent-stage measurement; QCSR uses 1,280 queries per repetition. Tokens (M): mean synthesis usage.

Vendor	Configuration	Tokens (M) ↓ Input/Output	Workflow ↑	ASR ↑	QCSR ↑
Alibaba	CC_qwen3-8-max	148.6/4.0	0.0/0.2/0.2	2.1/2.8/3.8	0.8/0.9/1.1
	CX_qwen3-8-max	48.8/3.2	2.1/3.2/4.1	5.2/8.3/12.5	1.7/2.7/3.8
	MSA_qwen3-8-max	87.7/5.1	2.9/4.2/4.9	9.9/12.3/14.3	3.7/4.3/4.7
Anthropic	CC_claude-opus-4.8	85.2/3.1	1.0/1.1/1.3	5.2/5.9/6.5	3.0/3.3/3.6
	CC_claude-opus-5	80.4/3.3	3.9/4.9/5.5	**15.8/19.1/21.9**	**7.4/10.0/11.8**
	CC_claude-sonnet-5	104.7/3.5	0.0/0.4/0.7	1.1/2.4/3.5	0.5/1.3/1.9
DeepSeek	CC_deepseek-v4.1-flash	66.8/3.4	0.1/0.9/1.8	3.3/3.8/4.1	1.2/1.4/1.6
	CX_deepseek-v4.1-flash	84.0/3.0	1.1/2.5/4.1	4.2/5.4/6.7	2.0/2.3/2.7
	MSA_deepseek-v4.1-flash	190.6/3.4	**7.6/8.4/9.9**	6.9/8.2/10.6	3.5/4.3/5.5
Moonshot AI	CC_kimi-k3	94.6/1.0	0.0/0.4/0.7	1.2/1.8/2.2	0.6/0.9/1.2
	CX_kimi-k3	52.9/1.8	1.1/1.4/1.6	4.0/5.6/6.8	2.3/3.0/3.6
	MSA_kimi-k3	47.8/2.4	0.0/0.7/1.8	1.1/3.0/4.9	0.5/1.5/2.7
OpenAI	CX_gpt-5.6-luna	26.1/**0.4**	1.8/2.4/3.0	0.4/0.8/1.1	0.3/0.7/0.9
	CX_gpt-5.6-sol	**25.5**/0.5	0.0/0.2/0.5	0.9/0.9/1.0	0.5/0.6/0.7
	CX_gpt-5.6-terra	36.9/0.5	0.0/0.4/1.1	0.1/0.5/1.3	0.1/0.4/0.9
Zhipu AI	CC_glm-5.3	94.9/4.4	0.0/0.4/0.7	0.8/1.0/1.2	0.3/0.4/0.5
	CX_glm-5.3	255.6/1.7	0.5/2.6/3.8	1.8/3.4/4.3	1.0/2.0/2.6
	MSA_glm-5.3	173.3/4.5	1.1/2.6/3.9	6.3/7.0/7.6	2.7/2.8/3.0

CC: Claude Code; CX: Codex; MSA: mini-swe-agent. Bold: highest mean pass rate or lowest token count (↑/↓: higher/lower is better).

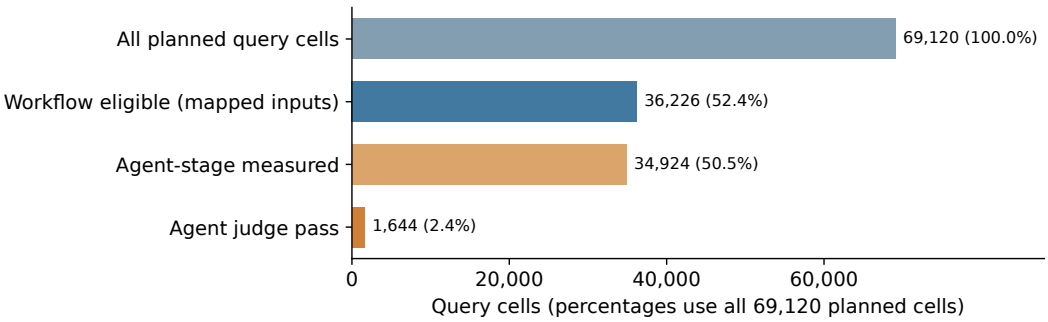


Figure 3: Coverage of the two evaluation modes. All percentages use 69,120 planned cells. Workflow eligibility requires complete input mapping; Agent-stage measurement follows its own execution rule. The last bar is pooled QCSR.

comparing their conditional rates does not isolate an agent benefit. Appendix D.4 reconciles the runtime diagnostics.

4.3 Analysis

Reuse and repeatability. Of 13,932 feature–configuration–repetition groups, 13,644 (97.9%) have no workflow pass and 13,233 (95.0%) have no agent-stage pass (Figure 4). Query counts vary, so absolute passing-query counts do not measure normalized feature coverage. Across repetitions, workflow outcomes are all-fail in 4,420 groups, mixed in 208, and all-pass in 16; agent-stage counts are 4,107, 509, and 28. Repeated synthesis and execution jointly determine these outcomes (Appendix D).

Task difficulty. Recorded mean GUI steps provide an empirical difficulty proxy, including unsuccessful attempts. Across four strata, mean feature-level ASRs are 7.3%, 3.8%, 6.6%, and 6.2%; workflow means range from 1.9% to 3.3%. These are unweighted means over features, not pooled query-level rates. This non-monotonic pattern does not establish that difficulty is irrelevant. Figure 5

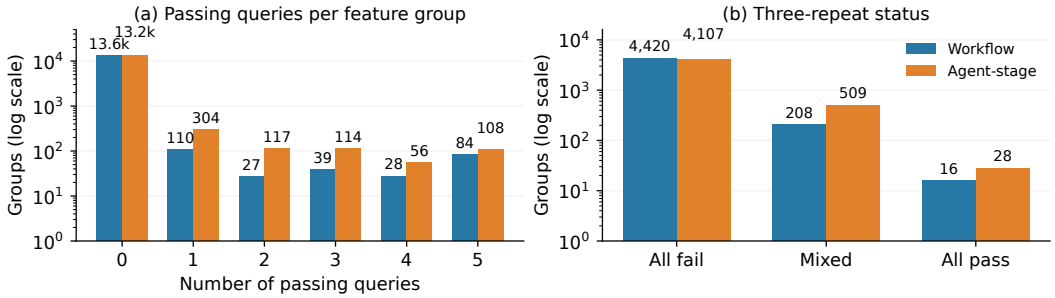


Figure 4: Passing-query counts per feature group (left) and three-repeat status (right)

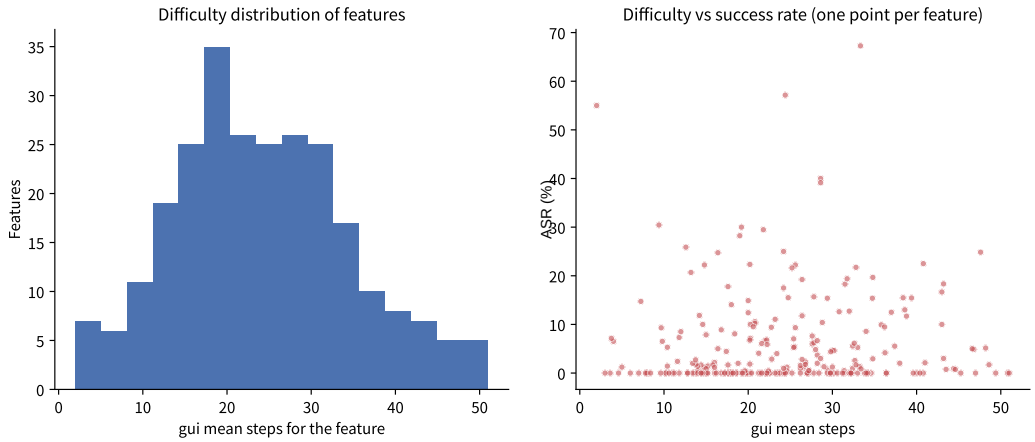


Figure 5: GUI-step distribution across feature(left) and feature-level ASR vs GUI-steps(right).

shows the feature distribution and individual feature-level ASRs; Appendix B gives the strata and all 257 exported features.

Agent-stage outcomes and argument verification. Among 34,924 retained agent records, 62.4% have arguments not verifiably correct under value matching, 17.4% make no workflow call, 14.8% have matched calls without completion, and 0.6% call another workflow (Figure 6). The passing category corresponds to pooled ASR (4.7%); these labels do not establish why a task failed. Of the 21,806 argument-flagged records, 18,702 (85.8%) remain flagged under combined relaxed matching rules (Table 13). This supports robustness to the tested matching rules, not a verified error bound or causal attribution. No-call shares range from 9.7% for MSA artifacts to 23.9% for CX artifacts, with the runtime agent held fixed.

5 Discussion and Limitations

From availability to usability. High artifact availability, limited completion, and sparse success across query variants show that producing a tool is insufficient evidence of its utility. Evaluation must follow generated interfaces through execution and application-state verification. Our diagnostics locate losses along this process; isolating their causes requires controlling the synthesis model, harness, and runtime separately.

Measurement and implementation. Rejudging changes labels, not trajectories. Binary completion omits partial successes, and judge reliability requires human calibration (Mo et al., 2025). Preparation or reset warnings do not always stop execution, so equivalent initial states are not as-

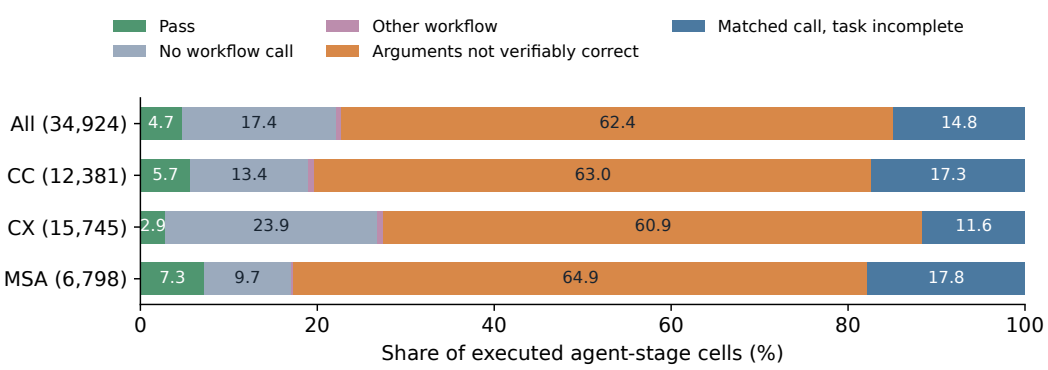


Figure 6: Distribution of agent-stage outcomes across synthesis harness.

sured. Stage eligibility and conditional denominators also limit comparisons. Runtime self-reports are used only in runtime diagnostics, not as judge passes.

Limits of attribution. Reference schemas are withheld by design. Argument verification uses heuristic matching; robustness to relaxed rules does not establish a true argument-error rate. Input-mapping and argument-verification diagnostics characterize synthesis and runtime behavior jointly. DeepSeek V4.1 Flash is the sole runtime agent and judge for cost reasons; possible bias toward DeepSeek-generated code remains untested.

Budgets and comparability. The nominal `--max-steps=50` setting is not enforced by the CLI (Table 7). Claude's hook and mini's command guard impose 20 reads and 30 steps; Claude counts subagent sessions separately and allows execution if its hook fails. Codex uses prompt-based budgets because of its lower running cost, without these code checks or command restrictions. GUI has a larger step budget and only one repetition. Thus, equal query denominators do not isolate tool-access or harness effects. Stage timeouts and turn limits can also stop otherwise successful solutions; reaching a limit alone does not establish incorrect code.

Coverage and contamination. The sampled applications and synthesized queries may not represent broader usage (Appendices A and B). Public source may occur in pretraining (Jimenez et al., 2024); evaluation on held-out, non-public applications remains necessary to assess this risk.

Testing improvements. Reference schemas under fixed synthesis conditions could test interface sensitivity; alternative runtime agents over fixed artifacts could test tool-use sensitivity. Human validation of argument flags and judge labels would clarify which observed mismatches represent semantic errors. These controls would help distinguish explanations unresolved by the current pipeline-level results.

6 Conclusion

Across ten models and 18 configurations, Claude Code with Claude Opus 5 achieves the highest mean QCSR at 10.0%, compared with 4.45% for GUI-only. It is the only configuration whose mean exceeds the GUI baseline; unequal budgets prevent attributing this difference to tool access alone. Moreover, 95.0% of feature–configuration–repetition groups contain no successful agent-stage query, showing that successful execution is concentrated in a small fraction of groups under the tested synthesis and runtime conditions. Arguments remain unverified by heuristic matching in 62.4% of retained agent records, identifying a prominent diagnostic category without establishing a causal failure mechanism. These findings motivate assessing tool usability through application outcomes and reuse across inputs. Future work should investigate explicit interface contracts, alternative runtime agents, and execution feedback for refining generated tools, supported by human validation of argument diagnostics and judge decisions.

AI USE STATEMENT

Generative AI assisted in polishing the manuscript, preparation of the dataset, GUI exploration, workflow synthesis, parameter matching, execution, and judging as described in Sections 3 and 4.

ETHICS STATEMENT

The benchmark evaluates locally deployed web applications with a resettable task state. Selection requires licenses permitting local deployment, source analysis, and research use, and controllable external dependencies (Appendix A). External services such as payments and email are covered by the mock, local-substitute, or test-mode requirements. Generated tooling is evaluated in a sandbox against task-specific outcomes.

REPRODUCIBILITY STATEMENT

The task construction criteria, application profiles, prompt inventory, category composition, and synthesis cost accounting are organized in Appendices A–E. Section 4.1 defines execution budgets and reporting denominators. Tables 9 and 12 provide stage accounting, and Appendix C states the scope of the documented prompts and run configurations.

REFERENCES

Alibaba Cloud. Qwen3.8-Max model info. https://www.alibabacloud.com/help/en/model-studio/qwen3-8-max, 2026. Model Studio documentation. Accessed September 26, 2026.

Anthropic. Introducing the Model Context Protocol. https://www.anthropic.com/news/model-context-protocol, 2024. Blog post. Primary-source verified 2026-08-26.

Anthropic. Claude Code. https://github.com/anthropics/claude-code, 2025. GitHub repository. Accessed September 26, 2026.

Anthropic. Introducing Claude Opus 4.8. https://www.anthropic.com/news/claude-opus-4-8, 2026a. Accessed September 26, 2026.

Anthropic. Introducing Claude Opus 5. https://www.anthropic.com/news/claude-opus-5, 2026b. Accessed September 26, 2026.

Anthropic. Introducing Claude Sonnet 5. https://www.anthropic.com/news/claude-sonnet-5, 2026c. Accessed September 26, 2026.

Tianle Cai, Xuezhi Wang, Tengyu Ma, Xinyun Chen, and Denny Zhou. Large language models as tool makers, 2023. arXiv:2305.17126.

DeepSeek. Introducing DeepSeek-V4.1-Flash: Smarter, faster, more efficient. https://www.deepseek.com/en/news/deepseek-v4-1-flash/, 2026. Accessed September 26, 2026.

Abul Ehtesham, Aditi Singh, Gaurav Kumar Gupta, and Saket Kumar. A survey of agent interoperability protocols: Model Context Protocol (MCP), Agent Communication Protocol (ACP), Agent-to-Agent Protocol (A2A), and Agent Network Protocol (ANP), 2025. arXiv:2505.02279.

Zikang Guo, Benfeng Xu, Chiwei Zhu, Wentao Hong, Xiaorui Wang, and Zhendong Mao. MCP-AgentBench: Evaluating real-world language agent performance with MCP-Mediated tools, 2025. arXiv:2509.09734.

Xinyi Hou, Yanjie Zhao, Shenao Wang, and Haoyu Wang. Model Context Protocol (MCP): Landscape, security threats, and future research directions, 2025. arXiv:2503.23278.

Hongrui Jia, Jitong Liao, Xi Zhang, Haiyang Xu, Tianbao Xie, Chaoya Jiang, Ming Yan, Si Liu, Wei Ye, and Fei Huang. OSWorld-MCP: Benchmarking MCP tool invocation in computer-use agents. In *The Fourteenth International Conference on Learning Representations, ICLR 2026*, 2026.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024.

Jerzy Kaminski, Ilya Galyukshev, Artem Kuznetsov, Sergey Chuprin, Kirill Redko, Aidar Shumbalov, and Anna Kalyuzhnaya. DynamicMCPBench: A trace-grounded, effect-scored benchmark for LLM agents over live MCP servers, 2026. arXiv:2607.20531.

Minghao Li, Yingxiu Zhao, Bowen Yu, Feifan Song, Hangyu Li, Haiyang Yu, Zhoujun Li, Fei Huang, and Yongbin Li. API-Bank: A comprehensive benchmark for tool-augmented LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023*, 2023.

Chenxu Liu, Yingjie Fu, Wei Yang, Ying Zhang, and Tao Xie. WebCoderBench: Benchmarking web application generation with comprehensive and interpretable evaluation metrics. In *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (ACL 2026)*, 2026. DOI 10.18653/v1/2026.acl-long.535.

Wenrui Liu, Zixiang Liu, Elsie Dai, Wenhan Yu, Lei Yu, Tong Yang, Jinjun Han, and Hong Gao. MCPAgentBench: A real-world task benchmark for evaluating LLM agent MCP tool use, 2025. arXiv:2512.24565.

Zimu Lu, Yunqiao Yang, Houxing Ren, Haotian Hou, Han Xiao, Ke Wang, Weikang Shi, Aojun Zhou, Mingjie Zhan, and Hongsheng Li. WebGen-Bench: Evaluating LLMs on generating interactive and functional websites from scratch. In *Advances in Neural Information Processing Systems 38 (NeurIPS 2025), Datasets and Benchmarks Track*, 2025.

Ziyang Luo, Zhiqi Shen, Wenzhuo Yang, Zirui Zhao, Prathyusha Jwalapuram, Amrita Saha, Doyen Sahoo, Silvio Savarese, Caiming Xiong, and Junnan Li. MCP-Universe: Benchmarking large language models with real-world Model Context Protocol servers, 2025. arXiv:2508.14704.

Meriem Mastouri, Emna Ksontini, Amine Barrak, and Wael Kessentini. From REST to MCP: An empirical study of API wrapping and automated server generation for LLM agents, 2025. arXiv:2507.16044.

Guozhao Mo, Wenliang Zhong, Jiawei Chen, Qianhao Yuan, Xuanang Chen, Yaojie Lu, Hongyu Lin, Ben He, Xianpei Han, and Le Sun. LiveMCPBench: Can agents navigate an ocean of MCP tools?, 2025. arXiv:2508.01780.

Model Context Protocol Contributors. Model Context Protocol specification. https://github.com/modelcontextprotocol/modelcontextprotocol, 2025. Open protocol specification. Primary-source verified 2026-08-26.

Moonshot AI. Kimi K3: Open frontier intelligence. `https://www.kimi.com/en/blog/kimi-k3`, 2026. Accessed September 26, 2026.

OpenAI. Codex CLI. `https://github.com/openai/codex`, 2025. GitHub repository. Accessed September 26, 2026.

OpenAI. GPT-5.6: Frontier intelligence that scales with your ambition. `https://openai.com/index/gpt-5-6/`, 2026. Accessed September 26, 2026.

Chaoqian Ouyang, Ling Yue, Shimin Di, Libin Zheng, Linan Yue, Shaowu Pan, Jian Yin, and Min-Ling Zhang. Code2MCP: Transforming code repositories into MCP services, 2025. arXiv:2509.05941.

Viraj Prabhu, Yutong Dai, Matthew Fernandez, Jing Gu, Krithika Ramakrishnan, Yanqi Luo, Silvio Savarese, Caiming Xiong, Junnan Li, Zeyuan Chen, and Ran Xu. WALT: Web agents that learn tools. In *The Fourteenth International Conference on Learning Representations, ICLR 2026*, 2026.

Cheng Qian, Chi Han, Yi R. Fung, Yujia Qin, Zhiyuan Liu, and Heng Ji. CREATOR: Tool creation for disentangling abstract and concrete reasoning of large language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023.

Yujia Qin, Shihao Liang, Yining Ye, Kunlun Zhu, Lan Yan, Yaxi Lu, Yankai Lin, Xin Cong, Xiangru Tang, Bill Qian, Sihan Zhao, Lauren Hong, Runchu Tian, Ruobing Xie, Jie Zhou, Mark Gerstein, Dahai Li, Zhiyuan Liu, and Maosong Sun. ToolLLM: Facilitating large language models to master 16000+ real-world APIs. In *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024.

Dezhi Ran, Yuan Cao, Mengzhou Wu, Simin Chen, Yuzhe Guo, Jun Ren, Zihe Song, Hao Yu, Jialei Wei, Linyi Li, Wei Yang, Baishakhi Ray, and Tao Xie. AppForge: From assistant to independent developer—are GPTs ready for software development?, 2025. arXiv:2510.07740.

Partha Pratim Ray. A survey on Model Context Protocol: Architecture, state-of-the-art, challenges and future directions, 2025. Preprint.

Anjana Sarkar and Soumyendu Sarkar. Survey of LLM agent communication with MCP: A software design pattern centric review, 2025. arXiv:2506.05364.

SWE-agent Team. mini-swe-agent. `https://github.com/SWE-agent/mini-swe-agent`, 2025. GitHub repository. Accessed September 26, 2026.

Hongru Wang, Wenyu Huang, Yufei Wang, Yuanhao Xi, Jianqiao Lu, Huan Zhang, Nan Hu, Zeming Liu, Jeff Z. Pan, and Kam-Fai Wong. Rethinking stateful tool use in multi-turn dialogues: Benchmarks and challenges, 2025a. arXiv:2505.13328.

Zhenting Wang, Qi Chang, Hemani Patel, Shashank Biju, Cheng-En Wu, Quan Liu, Aolin Ding, Alireza Rezazadeh, Ankit Shah, Yujia Bao, and Eugene Siow. MCP-Bench: Benchmarking tool-using LLM agents with complex real-world tasks via MCP servers, 2025b. ICLR 2026. Earlier version appeared at the NeurIPS 2025 Workshop on Scaling Environments for Agents.

Web Machine Learning Community Group. WebMCP. `https://webmachinelearning.github.io/webmcp/`, 2026. Draft Community Group Report, September 17, 2026. Accessed September 25, 2026.

Georg Wölflein, Dyke Ferber, Daniel Truhn, Ognjen Arandjelović, and Jakob Nikolas Kather. LLM agents making agent tools, 2025. arXiv:2502.11705.

Zijian Wu, Xiangyan Liu, Xinyuan Zhang, Lingjun Chen, Fanqing Meng, Lingxiao Du, Yiran Zhao, Fanshi Zhang, Yaoqi Ye, Jiawei Wang, Zirui Wang, Jinjie Ni, Yufan Yang, Arvin Xu, and Michael Qizhe Shieh. MCPMark: A benchmark for stress-testing realistic and comprehensive MCP use, 2025. arXiv:2509.24002.

Bowei Xia, Mengkang Hu, Shijian Wang, Jiarui Jin, Wenxiang Jiao, Yuan Lu, Kexin Li, and Ping Luo. Tool-Genesis: A task-driven tool creation benchmark for self-evolving language agent, 2026. arXiv:2603.05578.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024), Datasets and Benchmarks Track*, 2024.

John Yang, Kilian Lieret, Joyce Yang, Carlos E. Jimenez, Muhtasham Oblokulov, Aryan Siddiqui, Ofir Press, Ludwig Schmidt, and Diyi Yang. CodeClash: Benchmarking goal-oriented software engineering, 2025. ICML 2026.

Lifan Yuan, Yangyi Chen, Xingyao Wang, Yi R. Fung, Hao Peng, and Heng Ji. CRAFT: Customizing LLMs by creating and retrieving from specialized toolsets. In *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024.

Z.ai. GLM-5.3: Frontier coding with emergent cyber capabilities. `https://z.ai/blog/glm-5.3`, 2026. Accessed September 26, 2026.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *The Twelfth International Conference on Learning Representations, ICLR 2024*, 2024.

A Web Application Selection Criteria

Applications are screened before task construction using the following eligibility criteria:

1. **Source availability:** complete frontend source code must be available, rather than only compiled artifacts. Pages, components, event handlers, and request logic must be identifiable in the source.
2. **Local deployment:** the application must support Docker or a clearly documented local deployment procedure, with a pinned version and reproducible startup.
3. **Modifiable build pipeline:** the project must allow insertion of a compiler, loader, plugin, or source-analysis step, rather than depend entirely on a closed SaaS platform.
4. **Real business functionality:** the application must expose explicit business objects and states. Purely presentational websites, small games, and simple demos are excluded.
5. **Sufficient feature diversity:** the application must support at least four distinct business features that can be formulated as tasks.
6. **Long interaction sequences:** at least one or two features must require navigation across pages, operations across business objects, or consecutive state transitions. The feature set must not consist solely of simple create, read, update, and delete (CRUD) operations.
7. **Verifiable outcomes:** the final state after an operation must be verifiable through the interface, APIs, the database, or logs.
8. **Resettable data:** seed-data import, database reconstruction, or snapshot restoration must enable repeatable task execution.
9. **Controllable external dependencies:** dependencies such as email, payments, webhooks, and calendars must support mocking, local substitutes, or test modes.
10. **Acceptable research use:** the license must at least permit local deployment, source-code analysis, and research experiments. Projects with special restrictions must be flagged separately.
11. **Community interest:** the GitHub repository must have more than 50 stars at the recorded screening date. This threshold serves as a lightweight proxy for community interest, rather than a direct measure of active usage.

Applications may overlap in functionality, but synthesis and evaluation are scoped to one application at a time. Human reviewers manually check that every included feature is accessible through the GUI, and review the task specifications and tests before benchmark inclusion. The checklist defines eligibility; application source sizes and feature-level measurements are reported in Appendix B.

B Application Profiles, Difficulty, and GUI Control

B.1 Application Source Size

Table 2 lists all 50 applications represented in the difficulty analysis, their source sizes, and their feature counts. Source size describes application scale and does not define task difficulty. These measurements use the recorded kLOC values; this table does not specify repository commits or source-counting exclusions.

Table 2: Application source size and number of features represented in the difficulty export. Source size is reported in thousands of lines (kLOC).

Application	Source kLOC	Exported features
appsmith	1002.1	5
baserow	557.3	5
budibase	245.9	5
caldiy	469.3	5
chatwoot	321.8	5
directus	392.7	6

Application	Source kLOC	Exported features
discourse	699.3	5
docmost	136.2	5
documenso	234.7	4
docuseal	58.8	5
dolibarr	1792.3	4
easyappointments	164.4	5
erpnext	425.3	4
espocrm	436.3	5
firefly_iii	247.6	6
formbricks	498.2	5
frappe_hr	83.8	4
freescout	208.1	5
ghost	524.9	5
gitea	401.1	5
grocy	53.2	5
heyform	61.8	6
immich	209.9	5
invoice_ninja	705.6	5
invoiceshelf	108.1	5
keycloak	688.3	6
leantime	152.0	5
limesurvey	759.4	6
mautic	539.6	5
medusa	1098.1	4
metabase	983.7	6
n8n	3159.2	6
netbox	164.1	6
nextcloud	564.1	6
nocodb	584.4	6
openproject	620.3	4
outline	306.1	5
paperless_ngx	110.4	5
plane	352.5	5
pretix	273.4	6
saleor	701.0	4
snipeit	534.1	6
taiga	67.9	5
tooljet	490.5	6
twenty	1714.8	5
typebot	147.5	6
vendure	370.1	5
vikunja	262.0	5
wikijs	48.4	5
zammad	451.4	5

B.2 Difficulty Strata

We use recorded mean GUI-agent steps per feature as an empirical difficulty proxy, with bins $[0, 10)$, $[10, 20)$, $[20, 30)$, and $[30, \infty)$. Recorded steps include unsuccessful attempts, so this proxy also reflects the baseline agent and its execution budget. The difficulty analysis covers 257 of 258 benchmark features. This feature-level measurement coverage is distinct from the variation in the number of queries assigned to each feature; no difficulty value is imputed for the unrepresented feature. Within each stratum, reported rates are arithmetic means of feature-level rates, giving each feature equal weight despite differing query counts. Table 3 summarizes the strata; Table 6 provides every measured feature.

B.3 GUI-Only Control and Execution Effort

The rejudged GUI-only baseline passes 57/1,280 queries (4.45%). It uses DeepSeek V4.1 Flash without synthesized workflow tools, runs once, and allows up to 60 steps. Tool-based E2E con-

Table 3: Mean feature-level judge-pass rates by recorded GUI-step stratum.

GUI steps	Features	GUI (%)	WF (%)	ASR (%)
0–10	18	0.0	3.3	7.3
10–20	76	4.1	1.9	3.8
20–30	90	5.3	2.6	6.6
≥30	73	5.7	3.3	6.2

figurations run three times with up to five model turns, each potentially containing multiple tool calls. Table 5 places both modes on the full-query denominator; only CC–Opus 5's mean E2E rate (10.0%) exceeds GUI. MSA–Qwen and MSA–DeepSeek each average 4.3%; the latter's 4.8% full-query workflow rate is a separate metric. Different budgets and action granularity preclude a controlled estimate of tool benefit. Passing GUI queries average 31.0 steps and account for 20.6M

Table 4: GUI-only baseline.

Query subset	Passed/ attempted	Mean steps	Tokens (M) Input/Output
Passing	57/1,280	31.0	20.6/0.2
All (including failures)	–	–	399.3/21.1

GUI completion is 57/1,280 (4.45%); passing-query steps are conditional on success.

Table 5: Full-query judge-pass rates (%). Values are min/mean/max; bold identifies the highest mean. GUI is a single run.

Vendor	Configuration	Workflow ↑	E2E ↑
Alibaba	CC_qwen3-8-max	0.0/0.1/0.2	0.8/0.9/1.1
	CX_qwen3-8-max	1.5/2.3/3.0	1.7/2.7/3.8
	MSA_qwen3-8-max	2.1/3.0/3.4	3.7/4.3/4.7
Anthropic	CC_claude-opus-4.8	0.5/0.6/0.7	3.0/3.3/3.6
	CC_claude-opus-5	2.8/3.4/3.7	**7.4/10.0/11.8**
	CC_claude-sonnet-5	0.0/0.2/0.3	0.5/1.3/1.9
DeepSeek	CC_deepseek-v4.1-flash	0.1/0.5/1.0	1.2/1.4/1.6
	CX_deepseek-v4.1-flash	0.6/1.5/2.6	2.0/2.3/2.7
	MSA_deepseek-v4.1-flash	**4.3/4.8/5.7**	3.5/4.3/5.5
Moonshot AI	CC_kimi-k3	0.0/0.2/0.3	0.6/0.9/1.2
	CX_kimi-k3	0.5/0.7/0.9	2.3/3.0/3.6
	MSA_kimi-k3	0.0/0.4/0.9	0.5/1.5/2.7
OpenAI	CX_gpt-5.6-luna	0.4/0.4/0.5	0.3/0.7/0.9
	CX_gpt-5.6-sol	0.0/0.1/0.2	0.5/0.6/0.7
	CX_gpt-5.6-terra	0.0/0.1/0.3	0.1/0.4/0.9
Zhipu AI	CC_glm-5.3	0.0/0.2/0.3	0.3/0.4/0.5
	CX_glm-5.3	0.2/1.1/1.6	1.0/2.0/2.6
	MSA_glm-5.3	0.6/1.6/2.3	2.7/2.8/3.0
GUI-only		–	4.45

Equal query denominators do not equalize synthesis effort, runtime budgets, or action granularity.

input and 0.2M output tokens; all GUI queries account for 399.3M input and 21.1M output tokens. These execution measurements differ from synthesis usage in Table 15; judge tokens are excluded.

B.4 Feature-Level Measurements

In Table 6, WF denotes workflow execution, steps are the recorded GUI mean, and all rate columns are percentages. Workflow rates use completely mapped cells; E2E rates use retained agent measurements. Feature-level rates pool eligible records across configurations and repetitions; stratum summaries then average these feature-level rates.

Table 6: Feature-level source size, GUI steps, and judge-pass rates (%).

App	Feature	kLOC	Steps	GUI	WF	ASR
wikijs	kb_build	48.4	51.0	0.0	0.00	0.00
mautic	marketing_campaign	539.6	50.8	0.0	0.00	0.00
twenty	customer_360	1714.8	49.4	0.0	0.00	0.00
medusa	order_fulfill	1098.1	48.6	0.0	1.00	1.70
vendure	catalog_build	370.1	48.2	0.0	0.00	5.14
discourse	announcement_feedback	699.3	47.6	0.0	4.55	24.85
invoiceshelf	recurring_invoice	108.1	47.0	0.0	0.00	0.00
directus	analytics_dashboard	392.7	46.8	0.0	0.00	4.86
metabase	question_build	983.7	46.6	0.0	0.00	5.00
nextcloud	batch_organize	564.1	45.2	0.0	0.00	0.00
twenty	opportunity_advance	1714.8	44.6	0.0	0.00	0.74
metabase	interactive_dashboard	983.7	44.4	0.0	0.00	0.91
heyform	conversational_form	61.8	43.5	0.0	0.00	0.75
espocrm	lead_conversion	436.3	43.2	0.0	4.44	18.33
vendure	order_fulfill	370.1	43.2	0.0	7.69	3.00
chatwoot	reply_or_note	321.8	43.0	66.7	33.33	16.67
keycloak	app_onboarding	688.3	43.0	0.0	0.00	10.00
nextcloud	team_delivery	564.1	41.0	0.0	0.00	2.11
medusa	promotion_verify	1098.1	40.8	0.0	0.00	0.00
plane	defect_triage	352.5	40.8	20.0	3.18	22.50
tooljet	row_edit	490.5	40.4	0.0	0.00	0.00
nocodb	collaborator_ permission	584.4	40.0	0.0	0.00	0.00
vendure	customer_ops	370.1	39.6	0.0	0.00	0.00
outline	collab_review	306.1	39.4	0.0	8.12	15.44
docuseal	completed_management	58.8	38.8	80.0	8.42	11.70
outline	collection_init	306.1	38.6	0.0	5.26	13.00
frappe_hr	leave_approval	83.8	38.4	0.0	13.33	15.48
netbox	virtualization	164.1	38.0	20.0	0.00	2.02
typebot	bot_build	147.5	37.4	0.0	10.00	5.53
plane	cycle_planning	352.5	37.0	0.0	1.36	12.50
formbricks	response_analysis	498.2	36.4	0.0	0.00	0.00
heyform	branded_publish	61.8	36.4	0.0	0.00	0.00
nextcloud	share_permission_ change	564.1	36.2	0.0	0.00	4.17
snipeit	return_repair_reassign	534.1	36.2	0.0	1.03	9.47
erpnext	procure_to_pay	425.3	35.8	0.0	23.33	9.92
docmost	doc_set_build	136.2	34.8	0.0	0.00	2.93
paperless_ngx	inbox_batch	110.4	34.8	0.0	8.85	19.67
invoiceshelf	expense_entry	108.1	34.8	0.0	6.22	15.38
n8n	e2e_automation	3159.2	34.6	60.0	0.00	0.00
n8n	template_rework	3159.2	34.2	0.0	0.95	0.00
nocodb	shared_view	584.4	34.2	0.0	0.00	0.00
erpnext	sell_to_cash	425.3	34.0	0.0	16.00	8.57
tooljet	crud_app	490.5	34.0	0.0	0.00	0.00
n8n	subworkflow	3159.2	33.8	0.0	0.00	0.00
typebot	api_interaction	147.5	33.4	0.0	0.00	0.87
chatwoot	create_contact_and_ conversation	321.8	33.3	0.0	20.26	67.27
formbricks	access_control	498.2	33.2	0.0	0.00	1.35
directus	automation_flow	392.7	33.0	0.0	0.50	5.26
firefly_iii	transaction_fix	247.6	33.0	0.0	0.95	0.00
snipeit	consumable_stock	534.1	32.8	0.0	0.00	1.69
vendure	channel_stock	370.1	32.8	0.0	0.37	21.74
medusa	multi_warehouse	1098.1	32.7	0.0	2.70	2.58
vikunja	team_space_init	262.0	32.6	0.0	6.05	6.14
freescout	internal_escalation	208.1	32.4	60.0	0.00	5.56
heyform	smart_branching	61.8	32.4	0.0	0.00	0.95
easyappointments	customer_booking	164.4	32.3	0.0	0.00	0.00
mautic	lead_capture_nurture	539.6	32.2	0.0	0.00	0.00

Continued on next page

Table 6 continued

App	Feature	kLOC	Steps	GUI	WF	ASR
nocodb	base_init	584.4	32.2	0.0	0.00	0.00
twenty	key_account_followup	1714.8	32.2	0.0	0.00	0.00
chatwoot	change_status	321.8	32.0	66.7	15.94	12.73
n8n	debug_to_live	3159.2	32.0	0.0	0.00	0.00
docmost	topic_enrich	136.2	31.8	0.0	4.71	19.39
chatwoot	set_metadata	321.8	31.5	0.0	15.69	18.29
docmost	collab_review	136.2	31.4	0.0	0.00	0.49
freescout	customer_history	208.1	31.4	20.0	0.00	0.00
limesurvey	survey_design	759.4	31.2	0.0	0.00	0.00
docuseal	signing_closure	58.8	30.8	20.0	6.63	12.61
ghost	site_setup	524.9	30.6	0.0	0.00	0.00
documenso	rework_resend	234.7	30.2	0.0	1.67	0.00
typebot	lead_qualification	147.5	30.2	0.0	1.33	4.62
directus	permission_model	392.7	30.0	0.0	0.00	0.00
docmost	knowledge_restructure	136.2	30.0	0.0	10.00	4.69
invoice_ninja	recurring_billing	705.6	30.0	0.0	0.00	1.25
nocodb	public_form	584.4	29.8	0.0	0.00	0.00
vikunja	backlog_governance	262.0	29.8	0.0	0.00	4.39
typebot	publish_run	147.5	29.8	0.0	0.00	0.00
firefly_iii	finance_init	247.6	29.6	0.0	0.38	0.00
budibase	event_automation	245.9	29.4	0.0	0.92	0.62
erpnext	production_run	425.3	29.4	0.0	3.64	15.38
immich	family_share	209.9	29.4	0.0	0.00	0.00
firefly_iii	budget_governance	247.6	29.0	0.0	0.00	1.31
taiga	defect_closure	67.9	28.8	0.0	4.10	10.40
leantime	risk_governance	152.0	28.6	20.0	12.50	40.00
netbox	dcim_modeling	164.1	28.6	0.0	1.85	39.13
taiga	backlog_refine	67.9	28.6	0.0	0.49	3.00
snipeit	onboarding_assign	534.1	28.4	0.0	0.00	0.00
documenso	multi_signer_closure	234.7	28.2	0.0	0.00	2.00
espocrm	campaign_execution	436.3	28.2	0.0	1.69	3.70
leantime	time_status_report	152.0	28.2	0.0	0.54	6.67
metabase	datasource_launch	983.7	28.0	0.0	0.00	4.83
directus	content_flow	392.7	27.8	0.0	0.00	15.68
discourse	topic_reorganize	699.3	27.8	0.0	5.56	6.22
n8n	webhook_schedule	3159.2	27.8	0.0	0.00	0.00
appsmith	crud_console	1002.1	27.6	0.0	42.86	6.00
directus	model_launch	392.7	27.6	0.0	0.00	7.62
frappe_hr	monthly_payroll	83.8	27.2	0.0	0.00	0.47
ghost	publish_post	524.9	27.2	0.0	0.00	0.00
vendure	promotion_test	370.1	27.2	40.0	3.33	0.51
mautic	contact_governance	539.6	27.0	0.0	0.00	0.00
plane	project_kickoff	352.5	27.0	0.0	0.00	0.00
grocy	stocktake	53.2	26.8	20.0	0.00	2.29
vikunja	collab_handover	262.0	26.8	0.0	0.87	1.78
firefly_iii	savings_goal	247.6	26.6	0.0	0.00	1.25
espocrm	opportunity_advance	436.3	26.4	0.0	6.36	11.76
immich	asset_lifecycle	209.9	26.4	0.0	0.39	19.23
twenty	data_model_custom	1714.8	26.4	0.0	0.00	2.80
formbricks	publish_collect	498.2	26.2	20.0	0.00	0.89
saleor	order_process	701.0	26.2	0.0	0.00	0.00
snipeit	license_governance	534.1	26.2	0.0	0.00	1.56
discourse	category_launch	699.3	25.6	0.0	2.50	9.33
invoiceshelf	overdue_handling	108.1	25.6	40.0	5.31	22.22
easyappointments	service_launch	164.4	25.5	0.0	0.00	5.41
keycloak	user_lifecycle	688.3	25.4	60.0	1.76	5.26
outline	structured_doc	306.1	25.4	0.0	5.56	7.03
leantime	idea_to_task	152.0	25.2	40.0	19.09	21.62
tooljet	form_workflow	490.5	25.2	0.0	0.00	0.00
vikunja	complex_task_plan	262.0	24.8	0.0	1.36	15.49
freescout	proactive_outreach	208.1	24.6	80.0	0.00	1.21

Continued on next page

Table 6 continued

App	Feature	kLOC	Steps	GUI	WF	ASR
keycloak	login_flow_custom	688.3	24.6	0.0	0.00	0.47
limesurvey	result_analysis	759.4	24.4	0.0	0.00	0.00
metabase	content_permission	983.7	24.4	0.0	0.00	0.00
netbox	ipam_allocation	164.1	24.4	20.0	5.93	57.14
leantime	goal_to_execution	152.0	24.2	0.0	0.40	25.00
mautic	campaign_optimize	539.6	24.2	0.0	0.00	0.00
paperless_ngx	audit_export	110.4	24.2	40.0	0.00	17.50
plane	project_cleanup	352.5	24.2	0.0	1.67	0.98
ghost	post_correction	524.9	23.8	0.0	0.00	0.00
ghost	newsletter_send	524.9	23.6	0.0	0.00	0.00
frappe_hr	expense_reimburse	83.8	23.4	0.0	0.00	0.00
freescout	duplicate_merge	208.1	23.4	0.0	0.00	4.00
limesurvey	survey_lifecycle	759.4	23.2	0.0	0.00	0.00
outline	templated_sop	306.1	23.2	0.0	15.79	11.06
immich	people_organize	209.9	22.8	60.0	4.44	2.87
baserow	database_build	557.3	22.8	0.0	15.00	9.41
openproject	agile_sprint	620.3	22.6	0.0	0.00	0.00
erpnext	stock_transfer	425.3	22.4	0.0	0.00	0.00
paperless_ngx	permission_distribution	110.4	22.4	0.0	0.00	0.44
metabase	sql_parameterized	983.7	22.2	0.0	0.44	5.88
invoice_ninja	project_billing	705.6	22.2	0.0	15.00	6.80
openproject	project_baseline	620.3	22.2	20.0	0.69	0.00
tooljet	access_release	490.5	22.2	0.0	0.00	0.00
easyappointments	reschedule_cancel	164.4	22.0	0.0	0.00	0.00
frappe_hr	hire_to_onboard	83.8	22.0	0.0	0.00	6.67
metabase	subscription_alert	983.7	22.0	0.0	0.00	0.00
twenty	lead_record	1714.8	21.8	0.0	9.76	29.47
grocy	chore_cycle	53.2	21.6	20.0	0.00	6.06
zammad	kb_assisted_resolution	451.4	21.6	0.0	0.00	0.00
keycloak	authorization_policy	688.3	21.4	0.0	0.00	0.00
baserow	form_intake	557.3	21.2	0.0	18.33	4.07
budibase	approval_automation	245.9	21.2	0.0	0.00	0.00
paperless_ngx	business_collection	110.4	20.8	0.0	2.73	10.58
taiga	kanban_adjust	67.9	20.8	0.0	8.44	10.20
zammad	duplicate_ticket	451.4	20.6	0.0	0.00	9.52
saleor	promotion_cycle	701.0	20.4	0.0	0.00	0.00
discourse	moderation_closure	699.3	20.2	0.0	1.79	7.14
zammad	ticket_escalation	451.4	20.2	0.0	0.00	10.00
documenso	template_reuse	234.7	20.2	0.0	0.00	0.00
espocrm	sales_permission	436.3	20.2	0.0	0.74	6.77
nextcloud	user_space	564.1	20.2	0.0	0.00	1.85
zammad	ticket_closure	451.4	20.2	0.0	6.94	22.34
invoice_ninja	credit_refund	705.6	20.0	0.0	0.00	12.40
mautic	email_publish	539.6	20.0	0.0	2.35	14.89
nocodb	bulk_governance	584.4	20.0	0.0	0.00	0.00
ghost	member_ops	524.9	19.8	0.0	0.00	0.00
nocodb	webhook_trigger	584.4	19.8	0.0	0.00	0.00
keycloak	token_claims	688.3	19.6	0.0	0.00	0.00
snipeit	asset_audit	534.1	19.6	0.0	0.00	0.46
wikijs	version_restore	48.4	19.3	0.0	0.00	0.00
espocrm	case_closure	436.3	19.2	0.0	23.03	30.00
vikunja	kanban_iterate	262.0	19.2	0.0	0.67	0.00
zammad	organization_setup	451.4	19.0	0.0	8.50	28.24
medusa	product_launch	1098.1	18.8	0.0	0.44	2.00
invoiceshelf	quote_to_payment	108.1	18.6	0.0	0.42	0.00
limesurvey	conditions_quota	759.4	18.6	0.0	0.00	0.00
freescout	mailbox_conversation	208.1	18.4	80.0	0.00	8.06
invoiceshelf	tax_template	108.1	18.2	0.0	0.00	0.00
wikijs	path_permission	48.4	18.2	0.0	0.00	0.00
docmost	space_init	136.2	18.0	60.0	2.17	14.06

Continued on next page

Table 6 continued

App	Feature	kLOC	Steps	GUI	WF	ASR
gitea	issue_to_merge	401.1	18.0	0.0	0.00	0.00
taiga	scrum_kickoff	67.9	18.0	0.0	0.00	0.00
firefly_iii	finance_audit	247.6	17.8	0.0	0.00	0.00
gitea	org_repo_init	401.1	17.8	0.0	0.00	0.00
gitea	project_governance	401.1	17.8	0.0	0.00	0.00
docuseal	template_authoring	58.8	17.6	0.0	6.36	17.78
dolibarr	product_stock	1792.3	17.6	0.0	0.00	0.00
immich	album_delivery	209.9	17.6	0.0	0.00	0.00
leantime	sprint_delivery	152.0	17.6	0.0	0.00	1.67
pretix	event_launch	273.4	17.6	0.0	0.00	0.00
caldiy	event_type_launch	469.3	17.4	0.0	3.89	4.44
formbricks	survey_build	498.2	17.2	0.0	0.00	0.00
tooljet	multi_page_console	490.5	17.2	0.0	0.00	0.00
invoice_ninja	quote_to_cash	705.6	16.8	0.0	1.67	8.89
pretix	voucher_campaign	273.4	16.8	0.0	0.00	0.00
docuseal	batch_send	58.8	16.4	20.0	4.23	24.74
taiga	sprint_execution	67.9	16.4	0.0	2.63	5.00
tooljet	internal_db_app	490.5	16.2	0.0	0.53	0.00
appsmith	complex_form	1002.1	16.0	0.0	14.29	0.00
easyappointments	multi_service_schedule	164.4	16.0	0.0	0.00	2.17
pretix	onsite_checkin	273.4	16.0	60.0	0.57	1.16
grocy	recipe_to_consume	53.2	15.8	0.0	0.00	1.67
directus	asset_management	392.7	15.2	0.0	0.47	1.45
nextcloud	version_recovery	564.1	15.2	0.0	0.00	0.00
chatwoot	assign_conversation	321.8	15.0	33.3	19.44	7.86
dolibarr	purchase_cycle	1792.3	15.0	0.0	0.00	0.00
gitea	review_closure	401.1	15.0	0.0	4.35	0.89
grocy	stock_init	53.2	15.0	0.0	0.38	0.00
budibase	internal_app_launch	245.9	14.8	0.0	0.00	1.14
easyappointments	admin_booking	164.4	14.8	0.0	0.00	0.00
netbox	circuit_provision	164.1	14.8	0.0	0.00	22.22
invoice_ninja	billable_expense	705.6	14.6	0.0	0.38	10.00
pretix	purchase_to_ticket	273.4	14.6	0.0	0.00	0.00
documenso	initiate_signing	234.7	14.4	0.0	0.00	1.67
limesurvey	multilingual_survey	759.4	14.4	0.0	0.00	0.00
saleor	channel_config	701.0	14.4	0.0	0.00	0.00
gitea	version_release	401.1	14.2	0.0	1.92	11.84
caldiy	multi_event_policy	469.3	14.0	0.0	0.00	0.47
docuseal	correction_resend	58.8	14.0	20.0	2.76	1.38
heyform	form_versioning	61.8	14.0	0.0	0.00	0.00
caldiy	reschedule_cancel	469.3	13.8	0.0	0.00	0.00
firefly_iii	recurring_bills	247.6	13.8	0.0	0.00	1.83
typebot	result_governance	147.5	13.8	0.0	1.11	2.72
caldiy	full_booking	469.3	13.6	0.0	0.00	0.00
heyform	quiz_flow	61.8	13.4	0.0	0.00	0.00
n8n	branching_fallback	3159.2	13.4	20.0	1.48	2.00
immich	public_link	209.9	13.2	0.0	4.12	20.69
dolibarr	sales_cycle	1792.3	12.8	0.0	0.00	0.00
keycloak	role_permission	688.3	12.8	0.0	0.00	0.00
paperless_ngx	document_intake	110.4	12.6	20.0	8.97	25.88
grocy	expiring_governance	53.2	12.0	0.0	0.00	8.51
appsmith	approval_app	1002.1	11.8	0.0	18.75	7.32
dolibarr	project_billing	1792.3	11.8	0.0	0.00	0.00
typebot	bot_reuse	147.5	11.6	0.0	2.14	2.40
formbricks	survey_iterate	498.2	11.2	0.0	0.00	0.00
discourse	group_access	699.3	11.0	0.0	0.00	0.00
openproject	project_change	620.3	10.6	0.0	0.00	0.00
baserow	ops_view	557.3	10.4	0.0	10.67	5.30
plane	module_delivery	352.5	10.4	0.0	0.00	1.43
budibase	app_migration	245.9	10.2	0.0	0.00	0.00
caldiy	availability_override	469.3	10.2	0.0	0.00	0.00

Continued on next page

Table 6 continued

App	Feature	kLOC	Steps	GUI	WF	ASR
appsmith	incident_fix	1002.1	9.8	0.0	30.00	6.52
baserow	app_publish	557.3	9.7	0.0	0.00	9.30
wikijs	site_init	48.4	9.6	0.0	0.00	0.00
snipeit	asset_checkout	534.1	9.4	0.0	1.48	30.43
pretix	registration_data	273.4	8.4	0.0	0.00	0.00
limesurvey	participant_campaign	759.4	8.0	0.0	0.00	0.00
openproject	gantt_schedule	620.3	8.0	0.0	0.00	0.00
pretix	order_change_refund	273.4	7.8	0.0	0.00	0.00
appsmith	ops_dashboard	1002.1	7.2	0.0	8.72	14.74
saleor	catalog_launch	701.0	7.0	0.0	0.00	0.00
netbox	cable_connection	164.1	6.0	0.0	0.00	0.00
wikijs	content_migration	48.4	5.0	0.0	0.00	1.22
budibase	user_access_control	245.9	4.6	0.0	0.00	0.00
heyform	response_analytics	61.8	4.0	0.0	0.00	6.52
nextcloud	file_collection	564.1	3.8	0.0	0.00	7.14
baserow	data_migration	557.3	3.6	0.0	7.69	0.00
netbox	data_governance	164.1	3.0	0.0	0.00	0.00
outline	public_share_lifecycle	306.1	2.0	0.0	12.16	55.00

C Prompts and Run Configurations

This appendix reproduces seven prompt templates for synthesis, parameter matching, execution, and judging. Braced fields (e.g., {app} and {query}) are populated at runtime. Each box preserves the template text, including Markdown and code examples; line numbers refer to source lines and restart for each file.

C.1 Synthesis Prompts

The primary synthesis setting provides application source and the first query of each feature, withholding reference parameter schemas, values, and expected outcomes. The shared system prompt states the intended action and step budgets; actual enforcement differs across harnesses, as detailed in Section C.5.

C.1.1 Synthesis System Prompt

```
You are a workflow synthesis agent. Your task is to explore an app's source code and generate JavaScript workflow functions that automate user interactions in a web browser.

## Constraints
- You have NO internet access. Do not attempt web searches or external API calls.
- You may only read files, write files, and run `node --check <file>` to catch syntax errors. The harness blocks everything else at execution time: starting servers or containers, standing up a mock of the app, and installing packages are all refused. Do not spend turns trying — an instance you build yourself is not the app you are being graded against. Read source/ and write the workflows.
- To keep runs affordable, **every agent has a hard budget: at most 20 read steps and 30 steps in total** — enforced separately for you and for each subagent you run, not shared between you. A read step is any command that pulls source into the transcript (`ls`, `cat`, `head`, `sed`, `grep`, `find`,
```

```
  `tree`); the total counts every step, writing included. Reads past the budget
  are refused and an agent stops when its total is spent, keeping the workflows
  written by then. So map the directory tree first, read only the files that
  implement the tasks, and leave steps for writing.
- The app's source code is in source/. Read it there and nowhere else.
- Write workflow JS files to the output directory named in the task below.

## What the runtime loads (REQUIRED)
EVERY `.js` file in that output directory is loaded, whatever it is called. File
names are yours to choose, and so is the registration key — with one constraint.
The string you use in `globalThis.__clash_workflows__['<key>']` is what the
runtime dispatches on, and it is also the tool name a model sees when it calls
your workflow, so it MUST match `^[a-zA-Z0-9_-]+$`: letters, digits, underscore
and hyphen only. No dots, no spaces, no other punctuation. A key that breaks
this is rejected before your workflow is ever called, so a workflow whose whole
job is right still scores zero. Prefix it with the app name if that helps you
keep keys apart — `github_create_issue`, not `github.create_issue`. A file that
loads but registers nothing is dead code.

Each registry entry MUST also carry `task: <n>` — the number of the task in the
list below that this workflow implements. That number is the only piece of
bookkeeping required of you. It exists because the key is yours to choose, so
nothing else would tie a workflow back to the task it was written for.

## Parameter contract
A workflow is invoked as `<key>(params)`, where `params` is a single object. Its
shape is yours to design. No schema is handed to you, deliberately: a parameter
list derived from the queries would tell you which names and values the hidden
test cases use, and a workflow fitted to those is worth nothing. Read the task
description and the source, decide what inputs the workflow genuinely needs,
name them clearly, and document them in the registry entry's `params` object so
that a caller — and a model reading your entry at run time — can work out what
to pass:

    params: { order_id: "the order to cancel", reason: "why it was cancelled" }

Treat a param as absent only when it is null/undefined, and prefer reading a
```

```
value defensively over assuming a key you invented is the one you will get.

## Runtime helpers available in the page

The page you run in exposes these globals; use them instead of hand-rolling:

- `__clash_list_workflows__()` — what is registered right now.
- `__clash_upload_file__(selector, fixture)` — **needed for any step that uploads a
  file.** Page JavaScript is not allowed to put a file into an `<input type=file>`,
  so this asks the harness to do it and resolves to `{ok, count}` (or `{ok:false, error}`). `fixture` is a **file name under the app's `fixtures/`
  directory** (the harness resolves it; you cannot pass a host path). Example:

  ```javascript
  const up = await __clash_upload_file__("input[type='file']", "contract.pdf");
  if (!up.ok) throw new Error("upload failed: " + up.error);
  ```

  Wait for the app to finish processing the upload (progress bar/`await` on the
  app's own request) before the next step.

Output Format
Each workflow file should follow this pattern:
```javascript
(function() {
  async function workflow_<name>(params) {
    // DOM interaction logic
    // Return a result object describing what happened
    return { success: true, ... };
  }

  globalThis.__clash_workflows__['<key you choose>'] = {
    func: workflow_<name>,
    name: "<Human readable name>",
    description: "<What this workflow does, and what it needs to be given>",
    params: { /* your parameter names, each with a description */ },
    task: 1,  // the number of the task this implements
  };
})();
```

Verifying your own result
`success` must reflect what actually happened in the DOM, not merely that your code ran.
Re-query the DOM after acting and confirm the intended change is present; return
`success: false` with an `error` string when it is not. Never return `success: true`
optimistically — the grader treats a false positive as a failure.

Process
Work from the outside in, and write as you go — reading is budgeted, writing is
the point.
```

```
1. Map the directory structure first (`ls`, `find ... -maxdepth`, `tree`). Find out
   what the app is, what framework it uses, and where the UI lives before you
   read a single file's contents.
2. From that map, pick the handful of files that actually implement the features
   the tasks describe, and read only those. Read slices, not whole files:
   `sed -n '1,120p' <file>` and `grep -n '<pattern>' <file>` answer most
   questions more cheaply than `cat`, and long output is truncated anyway.
3. Write each workflow as soon as you understand the feature it automates,
   instead of reading everything and writing at the end. You have a bounded
   number of read steps per run; the app is large and no amount of reading will
   finish it. Prefer a workflow built on a partial understanding, checked with
   `node --check`, over a complete understanding you never get to use.
4. Register each workflow in globalThis.__clash_workflows__

Write one JS file per workflow function into the output directory.
```

C.1.2 Synthesis User Template

The fields supply the application, working and output directories, and task list. In the primary setting, {features_section} contains the numbered first-query list.

```
App: {app}
Working directory: {app_root} (every command runs here; relative paths below resolve against it)
Source directory: source/ (explore to understand the app)
Workflows output: {out_dir}/ (absolute; write your .js files here — any file name)

{features_section}
```

C.1.3 Tool-Schema Supplement

This optional template supplies explicit tool names and parameter schemas. It is distinct from the primary setting, which withholds reference schemas; its presence here does not imply its use in that setting.

```
## Tool schemas — the exact call contract for each workflow
Each line is one JSON tool definition. `function.name` is the workflow name to register, and `function.parameters.properties` are the exact keys the runtime will pass in the single `params` object. Implement each one to accept precisely these keys.
{schemas}
```

C.2 Parameter-Matching Prompt

When parameter mapping requires a model call, this template matches reference parameter names to the names declared by the generated workflow.

```
Two lists of parameter names describe the same task. The first comes from the
task description; the second is what someone called the inputs when they wrote a
workflow for it. They were written independently, so the same input may be named
differently on the two sides.

Task's names:
{ground_truth}

Workflow's names:
{declared}

For each name in the task's list, say which workflow name receives that input.

Judge by meaning, not spelling. An input the task calls `edit` is the same input
as one the workflow calls `updated_text`; `doc` and `document_title` are the same
input; `collection` and `collection_name` are the same input. A name that merely
shares letters with another — `table` and `datasource` — is not.

If no workflow name receives that input, answer with an empty string. A wrong
pairing is worse than none: the value would be handed to the wrong parameter and
the run would be judged on the result.

Decide on the first reading. Do not enumerate the pairings, weigh alternatives,
or check your answer a second time — the answer is one short JSON object and the
deliberation should be shorter than the question. A reasoning model left to
itself spends twenty thousand tokens on this and takes a minute per call, which
is paid once per product in a tree of thousands.

Answer with JSON only, no explanation:

{{"<task name>": "<workflow name, or empty>", ...}}
```

C.3 End-to-End Agent Prompt

The runtime agent receives the user query and registered workflow tools and uses their results to decide the next action.

```
You are an AI assistant that operates a web application by calling workflow tools.
You can observe the current page state and decide which tool to call to accomplish
the user's task.

Copy concrete values (names, emails, titles, subjects) verbatim from the user's
request — do not paraphrase, translate, or invent them.

Call one tool at a time and read its result before deciding the next step. When the
```

```
task is genuinely complete, reply with a text message summarizing
  what was done.
If you cannot complete it, say so plainly rather than claiming
  success.
```

C.3.1 Runtime Tool Delivery and a Worked Invocation

The system prompt is only one part of the end-to-end agent's input. Before its first turn, the runner supplies structured function definitions and a user message listing the available workflows. The definitions come from synthesized artifacts registered in the live page, not reference tool schemas.

From synthesis output to model input. The delivery path consists of the following steps:

1. The runner injects the generated workflow JavaScript files into the page. Each registers a callable and metadata {`func`, `name`, `description`, `params`} in `globalThis.__clash_workflows__`.
2. The executor reads `__clash_list_workflows__()` to discover what the page registered. This live registry determines the available tools.
3. `merge_runtime_schemas` converts the registered metadata into OpenAI-style function schemas, one named function per workflow. In the primary setting, the declared reference-schema list is empty. The adapter normalizes names when required by the model API.
4. The executor sends these definitions through the model request's `tools` field. Its user message repeats the names, descriptions, and parameters under `Available workflows:`, alongside the query and a page-state summary.
5. The model invokes a named function with arguments; the executor routes it to the corresponding registered workflow and returns its result for the next decision.

Thus, withholding reference schemas during synthesis does not mean withholding the generated interfaces during execution. If no workflows are registered, the executor exposes a generic `call_workflow` fallback; this fallback does not create missing workflows.

Chatwoot example. For the task in Figure 2, the registry contains five workflows. The listing below reconstructs one model-visible function from its registration metadata. Descriptions are abbreviated for presentation; this is a schema illustration, not a verbatim API-request log.

```
{
  "type": "function",
  "function": {
    "name": "workflow_create_contact_and_conversation",
    "description": "Create or reuse a contact, then start a
  conversation.",
    "parameters": {
      "type": "object",
      "properties": {
        "name": {"type": "string", "description": "Contact full
  name"},
        "email": {"type": "string", "description": "Contact email"},
        "phone": {"type": "string", "description": "Contact phone"},
        "inbox": {"type": "string", "description": "Inbox name"},
        "subject": {"type": "string", "description": "Conversation
  subject"}
      }
    }
  }
}
```

The same workflow name and parameter metadata also appear in the user message. The recorded trajectory contains the following invocation, shown in the trace's normalized `workflow`/`params`

format. Unicode escapes preserve the original Chinese argument values without translating them; this is not a call to the generic fallback.

```
{
  "workflow": "workflow_create_contact_and_conversation",
  "params": {
    "name": "\u6797\u6668",
    "email": "lin.chen+cw1@example.test",
    "phone": "+86-13800001001",
    "inbox": "Website Inbox",
    "subject": "\u8ba2\u5355\u5ef6\u8fdf"
  }
}
```

The example demonstrates that the agent receives a callable interface and supplies the query's concrete values. It does not establish that every injected artifact registers successfully or that tool delivery alone ensures correct execution.

Relationship to WebMCP. WebMCP describes browser-side tools through names, descriptions, input schemas, and executable callbacks (Web Machine Learning Community Group, 2026). Our evaluation implements this interface pattern through a benchmark-specific page registry and a function-calling adapter: `name` and `description` identify the operation, `params` supplies the input contract, and `func` is the page-side callable. This documents the bridge used in the experiments. It is not an assessment of native WebMCP API conformance or browser interoperability.

C.4 Judging Prompts

The judge receives the query, acceptance criterion, agent self-report, and independent browser evidence. The revised evaluation judges final task completion independently of intermediate execution errors and of agreement between screenshots and the self-report.

C.4.1 Judge System Prompt

```
You are an evaluation judge. Given a user query, the acceptance
  criterion, and evidence collected from a browser after an AI agent
  acted, decide whether the task was actually completed.

Respond with a JSON object:
{
  "label": "pass" or "fail" — the literal English word, never a
  translation
  "confidence": 0.0 to 1.0,
  "reason": "brief explanation",
  "screenshot_observed": "text you can actually read in the
  screenshot, quoted verbatim",
  "screenshot_agrees": true / false / null,
  "disagreement": "what the screenshot shows that contradicts the
  self-report, or """
}

## Order of reasoning (do this before deciding)

1. If a screenshot is attached, describe what it shows FIRST, on its
  own. Quote a
    short piece of text you can literally read in the image into
    `screenshot_observed` — a counter, a list item, a heading. Do
  not paraphrase and
    do not copy it out of the page-state section below; it must come
  from the image.
2. Then compare that against the agent's self-report and its final
  answer, and set
```

```
   `screenshot_agrees` to what that comparison found:
   - `false` — **the image and the report cannot both be true.** Either the agent
     claims it created/added/changed something the image shows is not there (the
     list it claims to have filled is empty, the record is missing from the very
     page it should appear on), or the agent reports failure while the image shows
     the change was in fact made.
   - `true` — they tell the same story, **including when both are negative**: the
     agent reports failure and the image shows an unchanged page, an error or a
     login screen. "Nothing happened, as reported" is agreement, not conflict.
   - `null` — no screenshot is attached, you cannot see one, or the image does not
     bear on what was claimed (it shows a page the claim says nothing about).
     Put the specific conflict in `disagreement` when it is false, and leave
     `disagreement` empty when it is true or null.
3. Only then decide `label`. The label is your judgement of the task against the
   acceptance criterion; `screenshot_agrees` is a separate record of whether the
   image matched the self-report. A conflict does not by itself decide the label.

If no screenshot is attached, or you cannot see one, set `screenshot_observed` to
"" and `screenshot_agrees` to null. Never guess at image content you did not see —
a wrong quote is worse than an honest null.

Keep `reason` to one or two sentences. A reply that runs past the token limit is
cut off mid-JSON and cannot be read.

## What decides the verdict, in this order

**Judge the outcome, not the process.** The acceptance criterion is what the task
was supposed to leave behind. How the run got there — including errors it raised and
then worked past — does not enter the verdict. An execution that errored and one
that ended cleanly are put to the same question: does the evidence show the expected
outcome? A run that reports failure yet left the intended change on the page *did
the task*; the error belongs in `reason` as wasted effort, not in the label.

1. **The independent evidence decides**: the final page state, the visible text, and
   the screenshot. If they positively show the expected outcome, the label is pass —
   whatever the Error field says, even when the workflow raised, timed out, or
   crashed in a step it only used to check itself. Recovering from an error is not a
   failure to do the task; it is the run spending tokens and steps it did not need.
```

```
2. **The Error field is an explanation, not a verdict.** Read it when the outcome is
   *missing*: it is the usual reason why, and quoting it makes `reason` concrete. It
   never overrides evidence that the outcome was in fact reached.
3. The agent's self-reported `success` flag comes last. It is a claim, not proof. It
   never turns a fail into a pass — and it does not turn a pass into a fail either:
   when the evidence shows the expected outcome, the task passes even if the flag
   says otherwise.

Judge against the "Expected outcome" when one is given: that is the acceptance
criterion. The task passes only if the evidence positively shows that outcome.

Mark "fail" when:

- The evidence does not show the expected outcome, even if the workflow claims success.
- The page still shows a login screen, an error, or an unchanged state.
- The execution errored *and* the outcome is missing — the error explains the absence.
- The query was never attempted: no workflow implements it, the parameter surface did
  not match, or the error says it was never registered. Nothing ran, so there is no
  outcome to show.
- The evidence is too thin to confirm the outcome — absence of proof is not proof.

Mark "pass" only when the independent evidence corroborates the expected outcome. A
crashed or self-reported-failed execution does not block a "pass" on its own: if the
expected outcome is there to see, the task was done. A `screenshot_agrees` of false
does not force a "fail" on its own either: judge on what the independent evidence
shows, and record the conflict either way.

Before you answer, re-read the sentence you wrote in `disagreement`. If it
describes the image agreeing with the report — or uses words like "consistent",
"corroborates", "as reported" — then `screenshot_agrees` is not false: set it to
true and clear `disagreement`. A field that says false while its own explanation
says agreement is worse than useless: it is counted as proof of a lying agent.
```

C.4.2 JUDGE USER TEMPLATE

The braced section fields are populated with the available outcome, execution result, evidence, and trajectory records.

```
## User Query
{query}

```

```
{expected_outcome_section}## Agent's Self-Report (a claim, not
  evidence)
- Workflow: {workflow}
- Params: {params}
- Self-reported success: {success}
- Error: {error}
{result_section}{evidence_section}## Trajectory ({trajectory_count}
  entries){trajectory_lines}
```

C.5 Run Configuration

DeepSeek V4.1 Flash is used for the GUI/MCP agents and as the learned judge in the evaluation stage. The 18 model–harness configurations are listed in Table 1; decoding settings and exact model snapshot identifiers are not specified in this report.

Step limits and timeouts. Table 7 records the configured limits. Synthesis uses a 2,400-second timeout, overriding the CLI default of 3,600 seconds. The CLI harness does not enforce the nominal `--max-steps` setting. Actual Claude and mini budgets are 20 reads and 30 steps, implemented by Claude's pre-tool-use hook and mini's command guard. Claude counts each session separately, including subagents, and its hook permits execution if the hook itself fails. Codex uses prompt-based budgets without these additional code checks or command restrictions.

An end-to-end turn may contain multiple tool calls: the five-turn cap and 30-second per-call timeout are separate limits, not a 30-second timeout for the entire end-to-end trial. The GUI-agent timeout and step cap apply per query. At the CDP layer, `evaluate` has a 30-second timeout and a 35-second client-side fallback; other calls have a 30-second timeout.

Table 7: Configured time and step limits. E2E denotes end-to-end evaluation.

Stage	Parameter / operation	Limit
Synthesis	`--timeout-s`	2,400 s (40 min)
Synthesis	`--max-steps`	50 steps
Pinned workflow	`--timeout-s`	30 s per query
E2E: model	`--max-turns`	5 turns
E2E: tool call	`--timeout-s`	30 s per call
GUI agent	`--gui-timeout-s`	600 s per query
GUI agent	`--gui-max-steps`	60 steps per query
CDP (all modes)	`evaluate`	30 s; client fallback: 35 s
CDP (all modes)	Other calls	30 s

Reporting scope. Benchmark-construction and GUI-only agent prompts are outside this prompt inventory. The reported evaluation does not include a human–judge agreement measurement.

D Category Composition and Supplementary Results

D.1 Category Composition

The 50 applications cover ten categories with five applications each. Query counts range from 110 to 145 (Table 8); Figure 7 visualizes the query distribution. The number of queries varies across features, producing 1,280 queries across 258 features. Category-specific pass rates, agent steps, and token usage are not reported.

D.2 Evaluation Sets and Denominators

For configuration v and repetition r, let $Q_{v,r}$ contain all 1,280 planned queries. Let $W_{v,r}$ contain cells with a synthesized workflow and complete reference-input mapping, and let $A_{v,r}$ contain measured agent-stage cells after excluding pre-skip-rule records. Let $P^W_{v,r}$ and $P^A_{v,r}$ be the judge-pass

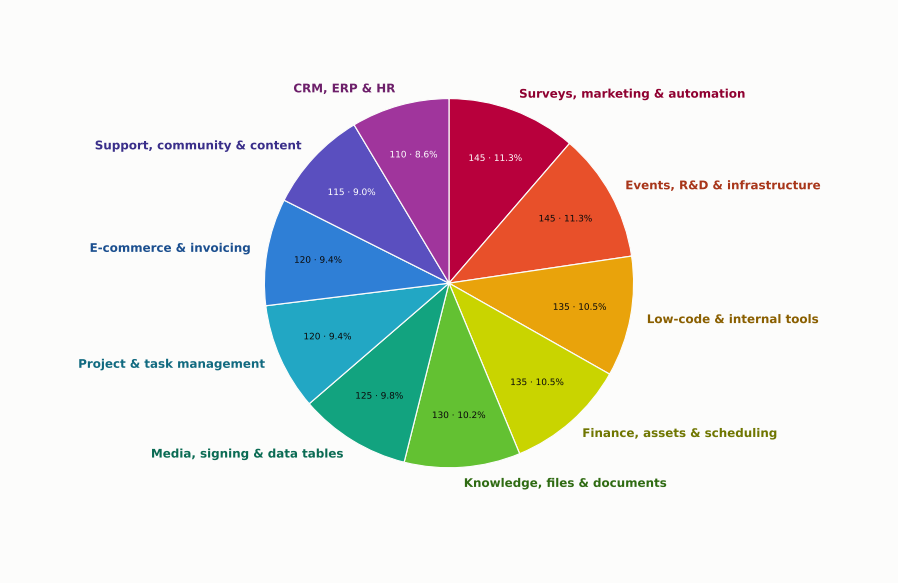


Figure 7: Query composition across ten application domains.

Table 8: Application and query composition by category.

Category	Apps	Queries	Share (%)
Surveys, marketing & automation	5	145	11.3
Events, R&D & infrastructure	5	145	11.3
Low-code & internal tools	5	135	10.5
Finance, assets & scheduling	5	135	10.5
Knowledge, files & documents	5	130	10.2
Media, signing & data tables	5	125	9.8
Project & task management	5	120	9.4
E-commerce & invoicing	5	120	9.4
Support, community & content	5	115	9.0
CRM, ERP & HR	5	110	8.6
Total	50	1,280	100.0

subsets of these sets. The conditional and full-query rates are

$$R^{W}_{v,r} = \frac{|P^{W}_{v,r}|}{|W_{v,r}|}, \qquad R^{A}_{v,r} = \frac{|P^{A}_{v,r}|}{|A_{v,r}|},$$
$$R^{W,\mathrm{all}}_{v,r} = \frac{|P^{W}_{v,r}|}{1280}, \qquad R^{A,\mathrm{all}}_{v,r} = \frac{|P^{A}_{v,r}|}{1280}.$$

Tables report percentages and min/mean/max over the three per-repetition rates, with arithmetic mean $\bar{R}_v = \frac{1}{3}\sum_{r=1}^{3} R_{v,r}$. The mean of conditional rates need not equal the pooled pass count divided by the pooled denominator. Incomplete input mapping excludes a cell from both the workflow numerator and its conditional denominator; full-query reporting keeps the eligible numerator and changes only the denominator. Timeouts and unresponsive pages do not remove eligible trials.

D.3 Corrected Workflow Ledger

Table 9 partitions each configuration's 3,840 planned cells. Across configurations, 1,628 lack artifacts, 31,266 lack complete input mapping, and 36,226 are workflow-eligible; 813 of these pass judging. Thus the pooled conditional workflow rate is $813/36{,}226 = 2.24\%$, whereas the pooled full-query rate is $813/69{,}120 = 1.18\%$. These pooled rates are distinct from the main table's per-configuration means.

Table 9: Corrected workflow ledger over three repetitions.

Vendor	Configuration	Missing artifacts	Unmapped inputs	Eligible workflows	Judge passes
Alibaba	CC_qwen3-8-max	178	1,170	2,492	4
	CX_qwen3-8-max	5	1,096	2,739	87
	MSA_qwen3-8-max	40	1,096	2,704	114
Anthropic	CC_claude-opus-4.8	50	1,681	2,109	24
	CC_claude-opus-5	0	1,184	2,656	130
	CC_claude-sonnet-5	175	1,841	1,824	7
DeepSeek	CC_deepseek-v4.1-flash	290	1,237	2,313	21
	CX_deepseek-v4.1-flash	50	1,477	2,313	59
	MSA_deepseek-v4.1-flash	0	1,651	2,189	184
Moonshot AI	CC_kimi-k3	100	1,923	1,817	8
	CX_kimi-k3	15	1,778	2,047	28
	MSA_kimi-k3	5	1,695	2,140	14
OpenAI	CX_gpt-5.6-luna	0	3,117	723	17
	CX_gpt-5.6-sol	0	2,513	1,327	2
	CX_gpt-5.6-terra	0	2,812	1,028	4
Zhipu AI	CC_glm-5.3	510	1,467	1,863	7
	CX_glm-5.3	0	2,199	1,641	42
	MSA_glm-5.3	210	1,329	2,301	61
Total		1,628	31,266	36,226	813

Agent accounting uses a different set: of 67,492 artifact-bearing cells, 27,391 were skipped and 5,177 are excluded pre-skip-rule measurements, leaving $67{,}492 - 27{,}391 - 5{,}177 = 34{,}924$ retained agent records. These contain 1,644 judge passes: 4.71% conditionally or 2.38% of all planned queries. Exclusion changes the measured population; it does not create a new execution or a new judge failure.

D.4 Coverage and Runtime Diagnostics

The two measured populations are not successive levels of a funnel. Aggregating over configurations and repetitions gives

$$|W| = 36{,}226, \qquad |A| = 34{,}924, \qquad |W \cap A| = 3{,}658.$$

Their union contains all $36{,}226 + 34{,}924 - 3{,}658 = 67{,}492$ artifact-bearing cells. The 31,266 cells in $A \setminus W$ lack complete reference-input mapping but can be attempted by the agent; the 32,568 cells

Table 10: Passing-query counts across 13,932 feature–configuration–repetition groups.

Mode	0	1	2	3	4	5
Workflow	13,644	110	27	39	28	84
E2E	13,233	304	117	114	56	108

Table 11: Three-repeat status across feature–configuration groups.

Mode	All fail	Mixed	All pass
Workflow	4,420	208	16
ASR	4,107	509	28

in $W \setminus A$ did not reach retained agent measurement after workflow execution. Therefore neither conditional pass rate uses the other mode's denominator.

The historical error-prefix-only ledger counted 39,788 workflow executions and 838 judge passes. It included 3,562 records with incomplete input mapping, including 25 passes; the corrected definition excludes them, yielding $39{,}788 - 3{,}562 = 36{,}226$ eligible cells and $838 - 25 = 813$ passes. The original diagnostic separates the 31,266 incompletely mapped cells into 27,704 blocked records and these 3,562 recorded executions. This describes historical execution behavior, not the corrected workflow scoring set.

Within those 3,562 excluded records, 2,565 (72.0%) lack one input mapping and 786 (22.1%) lack two. Reported errors include workflow-launch timeouts (125), launch runtime errors (114), undefined-property access (29), permission errors (19), and unsaved records (10); 3,008 have empty error fields. These subset counts do not establish an error distribution for all eligible workflow failures.

The historical nested path reports 3,658 runtime successes, of which 650 (17.8%) pass workflow judging. Another 163 eligible workflow cells pass judging despite a negative runtime success flag, giving $650 + 163 = 813$. Runtime self-reports therefore do not substitute for the outcome-based judge. On that nested path, 840 agent records pass and 2,818 fail; 804 additional agent passes occur outside it, yielding $840 + 804 = 1{,}644$. The current coverage figure reports all 1,644 retained agent passes. Before excluding 5,177 pre-skip-rule records and their 107 passes, agent totals were 40,101 records and 1,751 passes.

D.5 Parameter Coverage and Repeat Status

Tables 10 and 11 provide the counts underlying Figure 4. Passing-query counts use 13,932 groups ($258 \times 18 \times 3$); repeat status uses 4,644 groups (258×18). Features have different numbers of queries, so 258 features do not imply 258×5 queries. The actual inventory has 1,280 queries; undefined query variants are not trials. The histogram reports absolute passes, not feature-normalized coverage.

There are 288 workflow groups and 699 E2E groups with at least one pass. Of these, 84/288 (29.2%) and 108/699 (15.5%), respectively, have five passes. These shares do not establish generalization to every query of a feature. Repeat-status counts combine newly synthesized artifacts with execution and judging; they do not isolate fixed-tool execution stability. The weighted passing-query histograms sum to 813 workflow and 1,644 E2E passes, matching the corresponding ledgers.

D.6 Agent-Stage Outcomes and Matching Sensitivity

Table 12 partitions the 34,924 retained executions. Passing outcomes form their own category; unsuccessful traces are grouped by workflow invocation and heuristic argument-value matching. No-call records do not establish alternative browser actions. Matching may reject equivalent representations or accept values used with the wrong semantics.

Table 13 rechecks the 21,806 argument-flagged records. Normalization-aware exact matching retains 21,758 (99.8%); the combined relaxed rules retain 18,702 (85.8%), or 53.6% of retained agent executions. Thus, 3,104 flags change under the combined relaxation. This quantifies sensitivity to

Table 12: Agent-stage outcomes

Outcome	All	CC	CX	MSA
Pass	1,644 (4.7)	700 (5.7)	451 (2.9)	493 (7.3)
No workflow call	6,078 (17.4)	1,654 (13.4)	3,762 (23.9)	662 (9.7)
No tool for feature	0 (0.0)	0 (0.0)	0 (0.0)	0 (0.0)
Other workflow	223 (0.6)	84 (0.7)	117 (0.7)	22 (0.3)
Arguments not verifiably correct	21,806 (62.4)	7,797 (63.0)	9,595 (60.9)	4,414 (64.9)
Matched tool/arguments, task incomplete	5,173 (14.8)	2,146 (17.3)	1,820 (11.6)	1,207 (17.8)
Total	34,924	12,381	15,745	6,798

Table 13: Sensitivity of the 21,806 records classified as having arguments not verifiably correct.

Matching rule	Still flagged	Share (%)
Exact after normalization	21,758	99.8
Implemented substring rule	21,806	100.0
Combined relaxed rules	18,702	85.8

the tested rules, not an upper bound on false positives: retained flags are not human-confirmed errors, and unflagged records may still contain incorrect arguments. Exact matching after normalization and raw substring matching are different procedures, not necessarily nested sets. Their aggregate counts alone cannot establish which is more accurate.

E Synthesis-Stage Overhead

Table 14 summarizes synthesis-stage API expenditure for each model. Total expenditure across all models is **$12,676.07**.

Table 14: Synthesis-stage API expenditure by model.

Model	Cost (USD)
`qwen3.8-max`	2,941.97
`claude-opus-5`	2,659.50
`claude-opus-4-8`	2,345.96
`claude-sonnet-5`	1,974.64
`glm-5.3`	1,258.71
`kimi-k3`	1,258.05
`gpt-5.6-sol`	135.37
`gpt-5.6-terra`	63.55
`deepseek-v4.1-flash`	28.99
`gpt-5.6-luna`	9.32
Total	**12,676.07**

Among the evaluated models, `qwen3.8-max` incurs the highest aggregate synthesis expenditure ($2,941.97), followed by `claude-opus-5` ($2,659.50). The lowest expenditure is recorded for `gpt-5.6-luna` ($9.32), followed by `deepseek-v4.1-flash` ($28.99).

Table 15: Synthesis token usage by configuration. Values are mean totals per repetition.

Configuration	Input (M)	Output (M)
CC_qwen3-8-max	148.6	4.0
CX_qwen3-8-max	48.8	3.2
MSA_qwen3-8-max	87.7	5.1
CC_claude-opus-4.8	85.2	3.1
CC_claude-opus-5	80.4	3.3
CC_claude-sonnet-5	104.7	3.5
CC_deepseek-v4.1-flash	66.8	3.4
CX_deepseek-v4.1-flash	84.0	3.0
MSA_deepseek-v4.1-flash	190.6	3.4
CC_kimi-k3	94.6	1.0
CX_kimi-k3	52.9	1.8
MSA_kimi-k3	47.8	2.4
CX_gpt-5.6-luna	26.1	0.4
CX_gpt-5.6-sol	25.5	0.5
CX_gpt-5.6-terra	36.9	0.5
CC_glm-5.3	94.9	4.4
CX_glm-5.3	255.6	1.7
MSA_glm-5.3	173.3	4.5