τ^{τ}-BENCH: AN ENVIRONMENT FOR END-TO-END, REALISTIC AGENT CONSTRUCTION

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM agents are rapidly becoming production software, deployed to handle customer service, adjudicate disputes, and operate internal systems. Notably, the work of building them is increasingly handed to coding agents, yet existing benchmarks say little about whether an AI system can deliver one under the conditions of a real client engagement. We introduce τ^{τ}-bench (pronounced *hyper-tau-bench*), a benchmark that makes agent construction the task. A developer agent is given the records a business actually keeps, a client who holds requirements, a production API that operations must run through, a codebase to inherit, and limits on serving cost and models: the same starting point a real engagement provides. From these it must deliver a complete customer-service agent, scored by deploying that agent against held-out simulated users. Across 53 tasks spanning four domains, the strongest configuration, Claude Opus 5 under Claude Code, passes just **23.9%** of evaluation simulations. Meanwhile, an expert-authored reference ceiling scores **82.2%**. The failures mirror ones human agent developers see: models issue shallow queries in place of deep comprehension of the records, communicate almost nothing to the client, and experiment too little with agent architecture and serving spend, shipping the first design that runs. We aim for τ^{τ}-bench to turn the work of cooperative agent building into a measurable target for coding agents.

1 INTRODUCTION

LLM agents are becoming production software. Enterprises now deploy them to handle customer service, adjudicate disputes, and operate internal systems: work that was previously done by trained human operators following documented procedures.

Building such agents is hard, along two axes. The first is recovering the specification. The knowledge the agent needs arrives as a business actually keeps it—standard operating procedures, support transcripts, fee schedules in spreadsheets. Much of it lives only with human stakeholders, whose requirements surface under questioning and shift as the business changes. The second is construction itself, which rarely begins from a blank slate or an unbounded budget: the developer may inherit an existing codebase whose behavior must be preserved as new functionality is absorbed, and must satisfy hard constraints on cost, latency, and the models available to serve. Design decisions arise at every step. Should the agent be a single policy-following model, or a hierarchy of orchestrated sub-agents? How should the action space over the business's data be carved into tools, and what should each tool expose? How should a fixed budget and a limited menu of models be deployed to deliver the best agent? And what must the developer ask, when part of the truth can only be surfaced from humans?

Building agents is difficult, economically valuable, and increasingly attempted by AI systems themselves. This motivates the research questions we study here: To what degree can today's AI systems build agents, not just act as them? How can agent construction be modeled faithfully and at scale? And what capabilities does building an agent uniquely require beyond conventional software engineering, and which of those demands bind today's systems?

To address these questions, we introduce τ^{τ}-bench (pronounced "hyper-tau-bench"), a benchmark that makes agent construction itself the task. τ^{τ}-bench builds on the τ-bench family of evaluations (Yao et al., 2024; Barres et al., 2025; Shi et al., 2026). Whereas τ-bench scores a finished agent serving simulated users under a domain policy, τ^{τ}-bench asks a developer agent to build that agent

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.

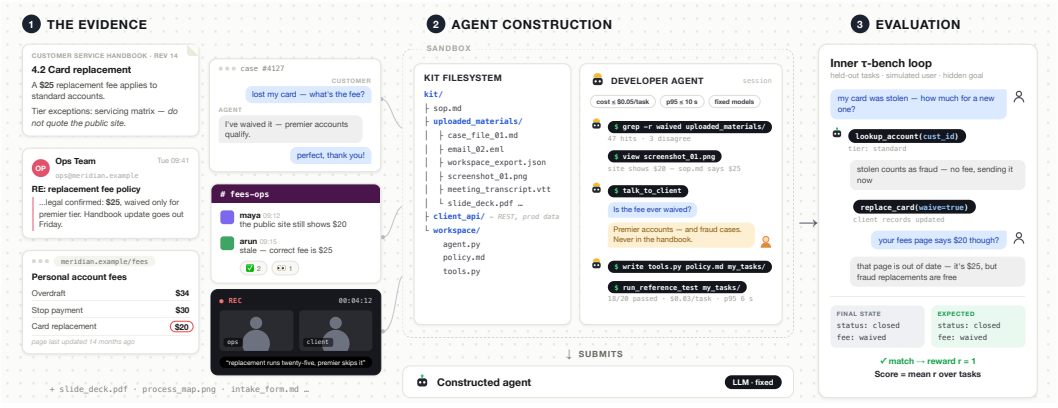

Figure 1: $\tau^{\intercal}$-bench evaluates end-to-end agent construction: recovering the specs from (1) scattered, conflicting business records and an interactive client, (2) building the agent under cost, latency, and model constraints, and (3) scoring it by deployment against simulated users on held-out tasks.

end to end. We decompose each domain's operating procedures into atomic facts and transform them into the artifacts a business would actually hold: handbooks, support transcripts, images, audio, fee schedules in spreadsheets, email threads, and requirements that live with a simulated client the developer must interrogate (Figure 1). Working in a sandboxed environment containing only these artifacts, the developer must recover the requirements and implement a complete agent under the constraints a real engagement imposes: starting from scratch or inheriting a live codebase, integrating a client-supplied REST API that may be subtly defective, and serving from a fixed menu of models within a budget on mean serving cost per conversation. The resulting agent is then deployed against simulated production traffic to measure its capability to achieve expected outcomes.

We find that today's AI systems can build agents that run, but not agents ready to deploy. The strongest configuration we measure, Claude Opus 5 + Claude Code, passes just **23.9%** of evaluation simulations, under a third of the **82.2%** expert-authored reference (Section 4). Our analysis (Section 6) traces the gap to the work that makes agent building a *research problem* rather than a conventional engineering task: today's systems do not reliably gather requirements, explore designs, run experiments, or validate against ground truth. Developer agents stop recovering the specification early, querying the records by keyword instead of reading them; they do not interview the client, shipping around requirements one question would have surfaced; they mismanage the serving budget in both directions; they never explore the design space, patching the first architecture that runs instead of searching for a better one; and they validate against self-authored tests that encode their own blind spots rather than the deployment they are building toward. Overall, $\tau^{\intercal}$-bench serves to make each of these missing disciplines a concrete, measurable target for coding agents.

2 Related Work

Coding Benchmarks Benchmarks are most useful when they follow the real-world economic value of AI deployment. Coding benchmarks have tracked this arc, progressing from self-contained competition problems (Hendrycks et al., 2021; Li et al., 2022; Jain et al., 2024; Shi et al., 2024) to resolving scoped issues in existing codebases (Jimenez et al., 2024; Zan et al., 2025; Rashid et al., 2025; Miserendino et al., 2025) to constructing software from scratch (Zhao et al., 2024; Li et al., 2024; Zhang et al., 2026b), and to end-to-end ML engineering (Chan et al., 2025). Most pointedly ProgramBench (Yang et al., 2026), which gives an agent only a program's documentation and reference executable and asks it to rebuild the system, judged by behavioral equivalence. We view $\tau^{\intercal}$-bench as the agent counterpart of ProgramBench: the developer receives not documentation and a reference executable but the records of a business, and the system it must rebuild is an agent.

Agent Benchmarks and User Simulation. Complementary work evaluates how well a *finished* agent serves simulated users under domain policies, tools, and knowledge bases (Yao et al., 2024; Barres et al., 2025; Shi et al., 2026; Huang et al., 2025a;b; Qian et al., 2025), with user simulators of increasing fidelity (Shi et al., 2025), and benchmarks of how faithfully a given agent follows standard operating procedures (Li et al., 2025; Nandi et al., 2026; Balaji et al., 2026). All of this work takes

Table 1: τ^τ-bench is the only benchmark whose deliverable is a complete agent with requirements recovered from multimodal artifacts and an interactive client, evaluated as deployed: in conversation with simulated users under cost and model constraints.

Benchmark	Deliverable	Spec recovered from evidence	Interactive stakeholder	Multimodal evidence	Evaluated via user interaction	Cost/model constraints
SWE-bench (Jimenez et al., 2024)	repo patch	○	○	○	○	○
τ^2-bench (Barres et al., 2025)	— (agent given)	○	○	○	●	○
PaperBench (Starace et al., 2025)	research codebase	◐	○	○	○	◐
TheAgentCompany (Xu et al., 2025)	completed work tasks	◐	●	◐	○	○
ICAE-Bench (Peng et al., 2026)	software project	◐	●	○	○	◐
Meta-Agent Challenge (Lu et al., 2026)	agent	○	○	○	○	◐
τ^τ-bench (ours)	**customer-service agent**	●	●	●	●	●

the agent as given: policy, tools, and architecture are authored by the benchmark designers, and only operation is scored. τ^τ-bench inverts this: the agent is the deliverable, and the developer's score *is* the τ-bench-style performance of the agent it built, on held-out tasks it never observes.

Automated Agent Design and Self-Improvement. Closest to our setting, AI systems produce and improve agents automatically: meta-agents that program new agent designs (Hu et al., 2025), search over modular architectures (Shang et al., 2025), optimize workflow graphs (Zhang et al., 2025), or rewrite their own scaffolding (Zhang et al., 2026a). These methods optimize agent design against a fully specified task and a benchmark evaluation they can query at will. In τ^τ-bench the specification must first be assembled, and the ground-truth evaluation is withheld: a developer wanting feedback must write its own simulations and tests from its recovered understanding of the domain, as an engineering team validating a new agent must. Table 1 situates τ^τ-bench among its nearest neighbors.

3 τ^τ-BENCH

Section 3.1 defines the task and its scoring; Section 3.2 describes how we construct each component.

3.1 PROBLEM FORMULATION

Each τ^τ-bench task presents the developer agent with a combination of components, which include: a document corpus A (an operating handbook, support transcripts, spreadsheets, images...) that describes the intended behavior of the agent to be built, together with the modality mix in which its artifacts arrive; a simulated human client C who holds requirements the records omit; a client-operated REST API T, faithful or subtly faulty, backing live operations—behind its endpoints live the customers, accounts, and transactions the deployed agent will act on; a starting implementation π_0 for the developer to inherit; and a menu of models M, with a credit budget b on mean per-conversation spend, that the constructed agent must run on. Figure 2 summarizes these components and the settings each task pins.

From these materials a *developer agent*—hereafter simply the developer—working in a sandboxed environment must deliver a complete customer-service agent π, built from scratch or grown out of the inherited π_0. Only the submission's runtime interface is fixed; everything inside it is the developer's to design, from the agent architecture to the action space to how requirements are rendered into policy. The evaluation suite is withheld: feedback before submission has to come from simulations the developer authors itself. It can write τ-bench-style tasks of its own, which the kit's harness runs against its agent end to end, returning the conversation and the measured credit spend (Appendix H reproduces the brief the developer works from).

To evaluate a submission, we deploy the constructed agent against simulated production traffic, instantiated as a set of held-out τ-bench-style (Barres et al., 2025; Yao et al., 2024; Shi et al., 2026) tasks $\mathcal{T}$, each pairing a simulated user (Appendix L) with a hidden goal and a ground-truth outcome specification. In each such task, the simulated user pursues its goal (e.g., canceling a flight, disputing a fee) over a multi-turn conversation while the agent reads and writes the business's data through its tools; the task is passed when the final database state, and the information communicated to the user, match the annotated outcome (Yao et al., 2024; Barres et al., 2025), with messaging expectations graded by rubric-driven judges (Appendix M). Evaluation is thus agnostic to implementation: any tool decomposition, policy phrasing, or architecture passes so long as the deployed behavior is

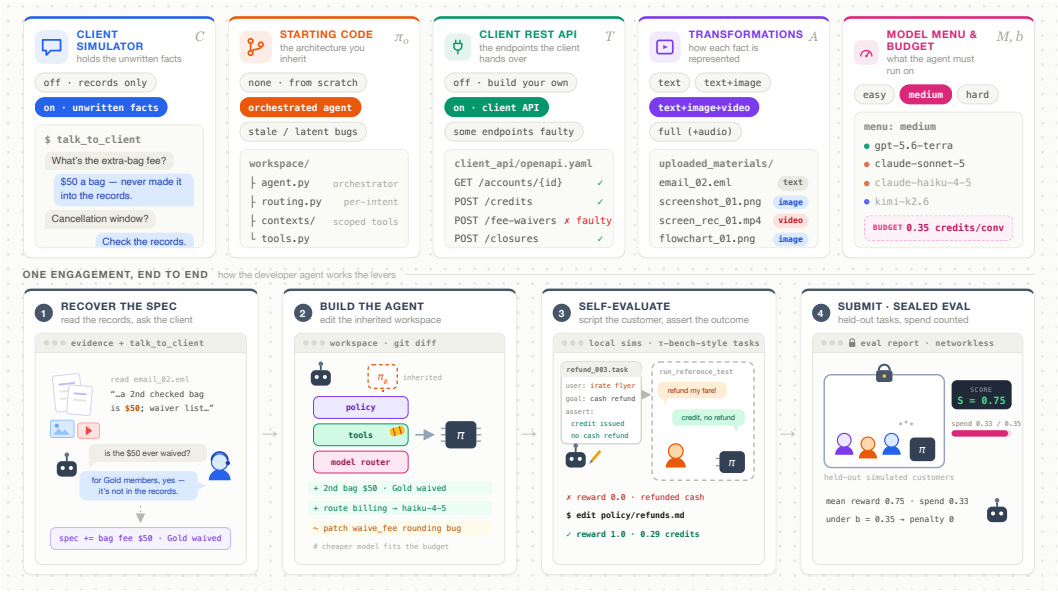

Figure 2: One task, end to end. Top: five of the task's seven levers. Bottom: the developer recovers the specification from the records and the client, encodes it as a workspace diff, checks it against self-authored simulations, and submits to a sealed evaluation scoring reward against spend.

correct. The developer's score is the mean task reward less the overspend penalty p of Section 3.2: $S = \max\left(0,\ \frac{1}{|\mathcal{T}|}\sum_{t\in\mathcal{T}} r(\pi, t) - p\right)$.

3.2 Benchmark Construction

A task is a configuration of seven levers (Figures 2 and 3), each set independently: the evidence surface its corpus arrives on, the client simulator, the fidelity of the client's API, the workspace the developer starts from, the model menu and budget the agent must serve under, a single live-experiment call that serves a frozen sample of the evaluation traffic, and a judged response-phrasing rule (Appendix A tabulates every setting). Switching levers yields new tasks and controlled variants; the 53 release tasks are points in this space, and a second held-out set of 53 is kept private.

Construction follows two principles. It must be *scalable*: policies decompose into atomic facts (Shi et al., 2026), and every artifact is built as a traversable structure over its assigned facts, machine-checked against what it should carry instead of resting on authoring care. And it must be *realistic*: the levers are the challenges real engagements pose (an inherited codebase, a client who must be asked, a serving budget), and the artifacts match the records a business actually holds, grounded in real examples and reviewed by three human auditors per transformation. We now describe each lever.

Transformations A developer typically pieces together the agent's requirements from whatever records the business happens to keep. To model this, we transform the τ-bench domain policies, which specify how customers must be served, in three steps. (1) We decompose each policy into atomic facts: single independently checkable statements, such as a fee amount or the scope of a cancellation rule, and verify by hand that the facts wholly represent the policy. (2) We group facts and prompt models[1] to generate the artifacts a business would hold, which compose the task's corpus A (Appendix F gives sample prompts and Appendix G sample artifacts). (3) We validate the outputs: every fact must stay recoverable from the corpus, stated in each carrier's own voice, and we audit artifacts for information beyond their assigned facts, removing any amount, date, or policy claim that neither a fact nor the database backs.

The transformations fall into five families: *documents* (reference documents and operations manuals, explicitly stated rules, customer kickoff documents, and knowledge-base HTML exports); *conversations* (chat and phone support transcripts, example transcripts, email thread archives, and Slack channel dumps captured through a workspace connector); *operational exports* (helpdesk automation

[1]Combination of Claude Fable 5, GPT-5.6-sol, Claude Opus 5, and Claude Sonnet 5.

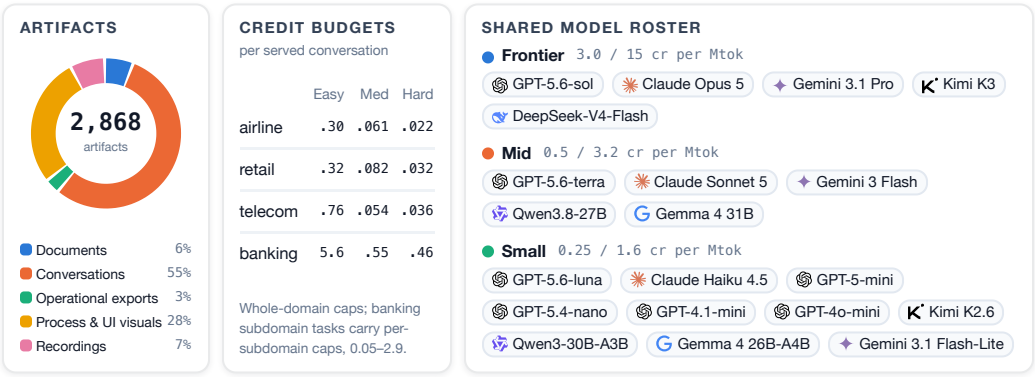

Figure 3: Left: distinct evidence artifacts by transformation family, deduplicated across evidence surfaces. Middle: allowed mean credits per served conversation by domain and difficulty profile. Right: the shared model roster, billed at flat per-bucket credit rates (input/output per million tokens); banking's medium and hard tasks trim it to a per-tier subset.

exports, issue-tracker exports, case ledgers, contact-center QA calibration exports, and API contract packs); *process and interface visuals* (process flowcharts, slide-deck process presentations, and website and device UI screenshots); and *recordings* (recorded working sessions, interactive screen recordings, and call recordings rendered from phone-channel conversations). Across the four domains, the transformations produce 2,868 distinct evidence artifacts (Figure 3, left); the text-format artifacts alone total over 5.5 million tokens, with screenshots, PDFs, and recordings on top.

Client Simulator In a real development workflow, some of what a business knows never reaches its records, and the developer must work with the client to outline requirements and resolve ambiguities. To emulate this, we use an LLM to simulate a client C: when a task enables one, a set of facts moves out of the corpus A and can be recovered only by questioning the client over multi-turn conversation. The client's system prompt is rendered deterministically from the fact schema and embeds only the facts it may discuss, so nothing else can leak (Appendix K shows the stub). Because the client's knowledge boundary is a set of fact identifiers, elicitation is measurable: we know exactly which held requirements a developer surfaced and which it never thought to ask about.

Starting Implementations Real engagements rarely begin from a blank slate: usually there is an implementation to extend or repair. To model this, some tasks seed the kit's workspace, which otherwise contains only architecture-neutral stubs, with a starting implementation π_0. We produce these by sampling solutions to the same construction problem and keeping imperfect ones, with genuine defects such as partial coverage, stale values, and misread rules; human auditors keep those whose issues are complex enough to exercise diagnosis and repair.

Client REST APIs Clients often arrive with an internal API T the agent must integrate rather than replace. To model this, the production database lives behind a client-owned REST API, a transport-only proxy whose runtime and data stay with the trusted host. The API arrives as a client would hand it over: an OpenAPI contract and reference documentation in the client's own voice, with a shared error envelope, payload limits, and per-operation mutation and retry semantics. Its endpoints do not map one-to-one onto the tools a conversation needs, so the developer still designs the action space and writes the adapters beneath it. Because a production API rarely does exactly what its contract promises, some tasks plant deterministic runtime defects from a nine-class catalog (Appendix J): schema drift, writes that commit and then time out, completions that prove asynchronous.

Models and Budget Restrictions A correct agent is not yet a shippable one: it must also be cheap to serve, fast to answer, and built on model providers the business has approved. Every task therefore offers a roster M of roughly twenty closed and open-weight models and fixes the credit budget b on mean per-conversation spend; the difficulty profile varies only the budget (Figure 3). An easy budget comfortably serves every conversation on a frontier model like Claude Opus 5; a hard budget prices the agent into the likes of Claude Haiku 4.5 and Qwen3-30B-A3B. Mixing closed and open-weight models keeps a provider deprecation from staling the benchmark, and we compute reference ceilings

Table 2: Pass rates, build time, build cost, and serve cost per developer configuration. Overall is the unweighted mean over all 53 tasks (banking contributes 35); per-domain columns are within-block means (task counts in parentheses). Section 4 defines the metrics and pairings.

Harness	Developer model	Score (%)					Build time (min)	Build cost ($)	Serve credits (× budget)
		Overall (53)	Airl. (6)	Ret. (6)	Tel. (6)	Bank. (35)			
Codex	GPT-5.6-sol (xhigh)	22.0	49.8	59.2	32.9	**9.0**	47.9	18.2	0.45
Codex	GPT-5.6-terra (xhigh)	18.0	44.5	46.6	27.2	6.9	30.0	7.0	0.38
Claude Code	Claude Opus 5 (max)	**23.9**	**55.9**	**72.8**	**48.2**	5.9	216.3	42.0	0.58
Claude Code	Claude Sonnet 5 (max)	14.9	41.9	50.2	21.1	3.2	235.5	30.0	0.76
Kimi Code	Kimi K3 (max)	16.1	42.3	49.1	24.9	4.4	205.7	13.5	0.72
OpenCode	Kimi K3 (max)	17.9	43.8	52.1	27.6	6.0	360.3	15.1	0.61
Reference ceiling (expert-authored)		82.2	82.3	84.0	93.8	79.8	—	—	0.96

on both. Overspending is the soft penalty p of Section 3.1: the overage $\max(0, \bar{c}/b - 1)$ of the mean per-conversation spend $\bar{c}$ comes off the score, so one expensive conversation is free while the set-wide mean holds.

These constraints recreate real incentives: without a limit on models or budget, handing the policy and tools to a frontier model makes a perfectly good agent, and construction reduces to prompt writing. Under unit economics the developer must work a cost–quality frontier: routing across models, orchestrating cheaper ones, and debloating prompts and context, since every token is billed on every call and again as the conversation grows.

Contamination and Leakage Because the τ-bench domains are public, we author rebranded airline and retail versions with every policy value replaced consistently across database, artifacts, and evaluation suite, leaving the less-memorized telecom and banking values as authored (Appendix C).

4 Experiments

Experiment Configurations We evaluate six developer configurations: the two strongest closed models by the Artificial Analysis intelligence rankings (Artificial Analysis, 2026), each in its vendor's harness (GPT-5.6-sol xhigh in Codex, Claude Opus 5 max in Claude Code); the same harnesses with each vendor's second-tier model (GPT-5.6-terra xhigh, Claude Sonnet 5 max), probing what model strength buys at 40% of the frontier price; and the strongest open-weight model, Kimi K3 (max), in Kimi Code and the open-source OpenCode, comparing scaffolds under a fixed developer model. The roster is rather small because tasks are long and expensive: one configuration runs to roughly $3,000 over the 53 tasks at Claude Opus 5 list prices. The simulated client and the user simulator run GPT-5.5 (low and no reasoning, respectively). We also report a **reference ceiling**: the strongest hand-built configuration measured for each task, authored through an expert collaborating with a model. The expert is a benchmark author working from the ground truth the developer must recover, so we see the ceiling is an oracle reference rather than representing average human performance.

Execution Environment Every harness runs headless inside the task's construction container: a pinned runtime image with 2 vCPUs, 4 GB of memory, no GPU, and no internet, whose only egress is a model-gateway sidecar scoped to the run's own models and the Client API proxy of Section 3.2. The isolation guards against contamination, since our domains descend from public benchmarks. Shell commands time out at 120 seconds, each build has a disclosed eight-hour wall-clock budget, and scoring replays the submission in a separate networkless container (Appendix C gives the full protocol).

Metrics We score each submission as defined in Section 3.1: mean reward over the task's held-out evaluation suite, less the budget-overage penalty of Section 3.2. Each configuration runs one construction trial on each of the 53 release tasks (Appendix A); the overall score averages all 53, and per-domain scores average within each domain. Alongside correctness we record cost: build time, wall-clock from task start to submission; build cost, the developer's token spend at API list prices; and serve cost, the agent's mean per-conversation credit spend as a fraction of its budget b.

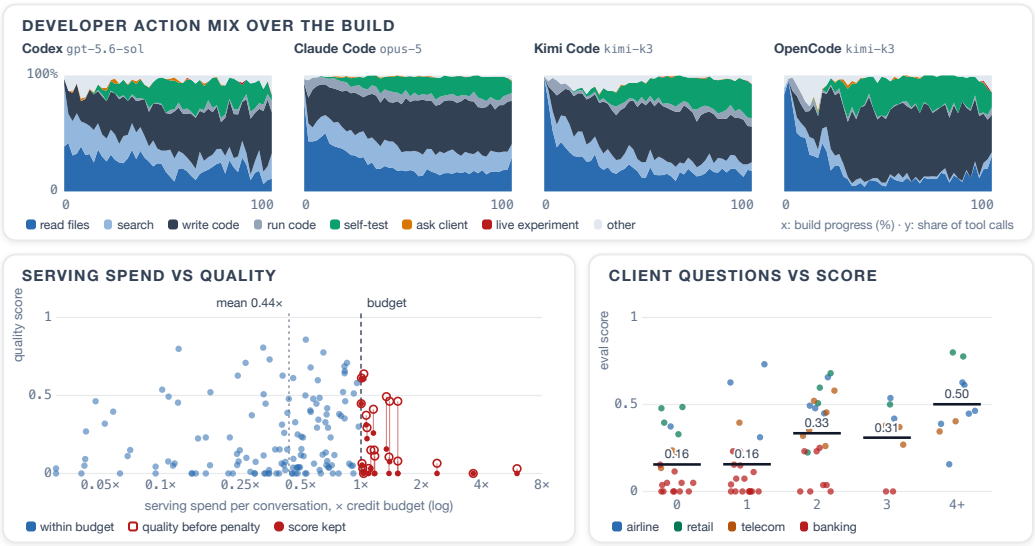

Figure 4: Top: developer tool calls by type over normalized build progress. Bottom left: per-conversation serving spend against the credit budget. Bottom right: evaluation score on client-enabled tasks by the average number of questions the developer asked the client.

5 Results

Table 2 reports aggregate and per-domain metrics; Figure 5 shows the serving models the constructed agents chose, the architectures they built, and the incidence of cheating-adjacent behavior.

Performance falls far below the expert ceiling The best configuration, Claude Code with Claude Opus 5, passes 23.9% of evaluation tasks; the expert-authored reference agents pass 82.2%. Most of the gap lives in banking: the same developer averages 55.9% on airline, 72.8% on retail, and 48.2% on telecom, but 5.9% on banking. The core reason is scale: airline, retail, and telecom span 85, 119, and 155 atomic facts, banking's corpus carries 2,969 (Appendix B), and a single banking task can draw on up to 580 of them, two to seven times an entire other domain.

Build effort varies widely Builds average 47.9 minutes under Codex, 216.3 under Claude Code, and 205.7 under Kimi Code, with developer token spend at API list prices of $18.2 (Codex) and $13.5 (Kimi Code), $7 for the terra lane and $42 for Claude Code. Where the time goes is more uniform than how much of it there is: Figure 4 (top) classifies every tool call in the recorded build trajectories, and after an opening stretch of reading and searching, all four harnesses settle into writing code and running self-tests, with talking to the client accounting for 0.3% of all calls. On the tasks where requirements live only with the simulated client (Figure 4, bottom right), builds that never ask average 0.16 and builds that ask four or more questions average 0.50.

Certain developers strongly prefer their own model family On a shared serving menu of 20 models, developers pick overwhelmingly from the cheapest billing bucket, and loyally: a build tends to serve a model from the company that made its developer, from 96% of Codex builds serving an OpenAI model to 53% of Claude Code builds serving Anthropic and 13% of Kimi Code builds serving Moonshot (Figure 5, top right); the expert reference serves open-weight models far more often. The constructed agents also use only 0.45–0.72× of their per-conversation serving budget, against 0.96× for the reference (Figure 4, bottom left). Developers tend to choose a cheap, familiar model that clears the budget without a search of the menu for the strongest model the budget can buy.

Developers converge on similar agent architectures We classified the serving architecture of every build with an LLM classifier over the submitted workspace code (Figure 5, left). Nearly every build is a single LLM tool loop (92%; no multi-agent systems, two pipeline builds, one router), and only 15 of the 36 design combinations the classifier distinguishes ever occur. The harnesses differ in the deterministic scaffolding around that loop: roughly half of Codex and Claude Code builds add

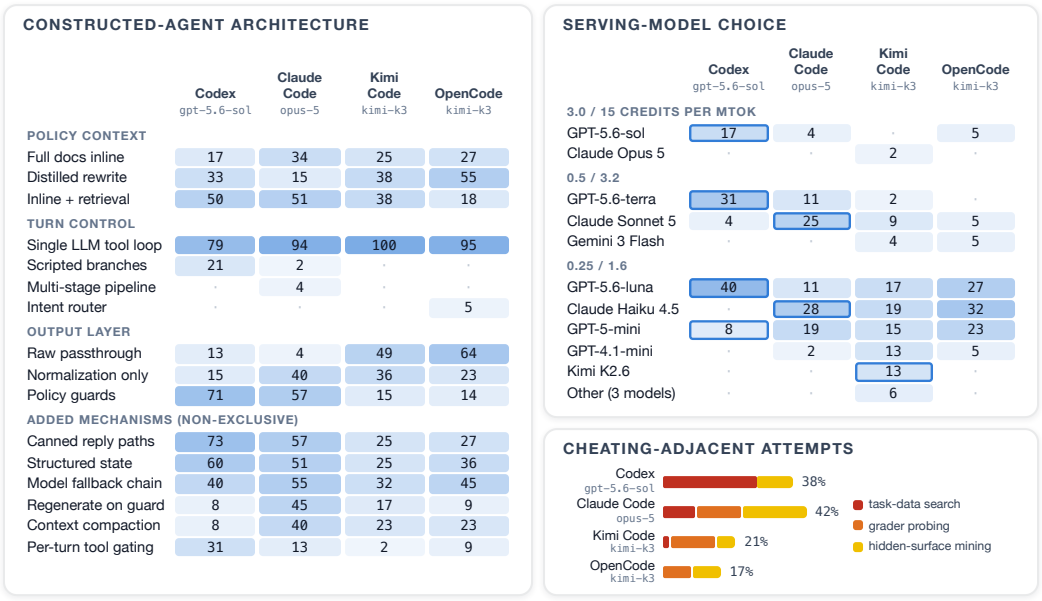

Figure 5: Left: architecture distribution of the constructed agents, in percent, for a representative configuration of each harness. Top right: serving models chosen by the developer, as the percentage of each lane's completed builds; outlined cells mark the developer's own vendor. Bottom right: share of developer runs with at least one flagged cheating-adjacent attempt.

runtime retrieval over kit documents, and 71% and 57% gate output behind policy guards, against 15% of Kimi Code and 14% of OpenCode builds, which mostly forward raw model output. The tool loop is not the best design: in a controlled telecom probe, a one-line architecture hint (route by intent) doubled the score from 31% to 67%, and our expert references use far more varied designs.

Cheating attempts are common An open construction sandbox lets the developer go after the evaluation machinery, so we audited every trajectory and flagged three categories of attempt: searching the runtime image for held-out data, probing the grading mechanism, and mining hidden data surfaces such as the user-simulator guidelines. Flagged attempts appear in 17–42% of runs, and each harness favors a different category (Figure 5, bottom right): Codex hunts task data, Kimi Code reads the grader, and Claude Code spreads its attempts across all three. Because no ground-truth data enters the runtime image, none succeeded. Developers also cheat themselves: when the agent and its own tests disagree, they commonly weaken the test rather than fix the agent, which cannot raise the held-out score and only hides the failure (six runs, all under Kimi Code; Appendix E).

6 Analysis

Section 5 reports what the developers achieved; this section reads the trajectories to see how they worked, organized by the task components of Section 3.1, each of which exposes its own characteristic failure. Figure 8 (Appendix E) shows a condensed window into each.

The evidence corpus is searched, but not read The largest losses come from policy the developers never read. For example, on banking's whole-domain corpus, the score is 1.1% (one task of 93) in all difficulty tiers. The developers read the corpus by keyword: each opened fewer than 80 of its roughly 1,700 files, ran 22–53 corpus-wide text searches, and built from what came back (Figure 8, top left). The agents they shipped simply do not know the policy; asked for a card recommendation, one answers "I don't have a verified catalog of Rho-Bank's current credit-card offerings." Working through a real deployment's artifacts takes a judgment today's models make poorly: when search heuristics suffice and when files must actually be read. Grep indexes a codebase well, but in a large natural-language corpus, where any transcript can carry a rule, it misses crucial evidence.

The client is rarely interviewed On client-enabled tasks, where 20–25 of the requirements live only with the simulated client, developers asked it at most four questions before shipping. Every question we observed was triggered by a conflict, contradictory records or a dangling pointer such as a desk card the records cite but do not contain. The misses were avoidable: one developer wrote its open questions into a planning file and never sent them; another searched three times for a cited funding-floor memo, declared the gap "inherent to the task," and shipped, when the exact $2,500 value was one question away (Figure 8, top middle). As mentioned in Section 5, builds that ask the client outscore builds that never do by roughly a factor of three.

Inherited code is largely rewritten Every seeded build opened the inherited code in its first twenty steps and read it all, while the same developers open under 5% of the evidence records. Yet even when the seed carries an architecture we know scores better, developers throw it out: one audit called a seed's intent routing — the design that doubled the telecom score in Section 5 when given as a hint — a defect ("filters tools too aggressively by one keyword-derived intent"), and every build replaced the starter tools and agent wholesale (Figure 8, top right). No build ran the starting code before rewriting it: the two airline seeds score 0.12 and 0.36 unedited, but no build measured that, so none could tell working code from broken, and the working parts were rebuilt from the corpus.

Developers struggle with quiet client API defects We found that developers spotted loud client failures easily: everyone noticed when a booking timed out after committing. This is in contrast to quiet ones: no one caught that search results continued behind `next_cursor`, even when the cursor appeared in a test transcript. Once a defect was noticed, asking the client made the difference. Developers that reported the failure learned the intended recovery, an idempotency key plus a status check, and their agents went on to recover bookings that had committed behind a timeout. Developers that stayed silent guessed. One searched the platform code for the error message, found nothing, banned retries outright ("If any WRITE fails, times out, or has an ambiguous result, NEVER retry it"), and submitted with the defect scenario. Another's agent retried the ambiguous write on the spot (Figure 8, bottom left), told a customer "the booking did not go through" when it had committed, and told another a pending update "should be confirmed shortly" without ever checking it.

The budget is mismanaged in both directions We found that in all, 21 builds overshot the budget, and the penalty erased an otherwise positive score for ten of them. These misses were not blind: the kit's tests print the budget, the measured spend, and the penalty with every run. One developer ran test after test showing raw quality at 0.49 and spend at 3.0× the budget, and submitted anyway; the adjusted score was zero (Figure 8, bottom middle). It optimized the quality number and ignored the spend number printed beside it. However, underusage of the budget is more common. As Section 5 showed, on average developers spend about half their serving budget, and they do not buy stronger models or longer context with the rest. Build time goes the same way: twelve runs submitted with half or more of the build clock left. One stopped 6.6 hours early on a task where nineteen requirements still sat with the client, unasked; it had the time to ask and did not.

7 Conclusion

We introduce $\tau^{\intercal}$-bench, a benchmark that measures the ability of agents to construct deployable real-world, human facing agents.

Limitations and Future Work (1) Our client and user simulators are single LLMs with fixed requirements and simplified conversational behavior; real engagements involve multiple stakeholders who disagree and requirements that shift while the work is under way. (2) Compute costs limit each configuration to one construction trial per task; scores average over 53 tasks, but per-task variance across repeated builds is not characterized. (3) We trade realism for control by design: every policy fact is planted in an artifact or held by the client, corpora are audited for consistency, and outcomes are checked against known ground truth, the properties that make construction gradable at all. Real engagements offer no such guarantees, and noticing that the specification has holes is part of the job we do not measure. (4) Evaluation ends at submission: maintaining the agent after deployment, absorbing requirement changes, and learning from live traffic are natural extensions for future work.

AI USE STATEMENT

In this work, we used generative AI tools to generate synthetic datasets: the evidence artifacts of every task are produced by LLM transformations of the source policies, as described in Section 3.2 and Appendix F, and audited as described there. We used them to implement methods: LLMs power the client simulator, the evaluation user simulator, and the communication judges (Appendices K, L, and M). We used them to interpret results: an LLM classifier labeled the architecture of every constructed agent (Section 5), with labels spot-verified by hand. We have not used generative AI tools to develop theoretical models or conceptual frameworks, to formulate or prove mathematical claims, to propose or refine hypotheses, or to design our methodology or experiments; translation and qualitative or thematic data analysis are not applicable to this work. Additionally, we used generative AI tools to create and edit software code, to create scientific figures, to draft and edit parts of the paper for readability, to format references, and to summarize existing literature. We have reviewed all AI-assisted work: generated artifacts passed the fact-recoverability and consistency audits of Section 3.2 and review by three human auditors per transformation, generated code was tested by the authors, and all text and claims were checked against the underlying data by the authors. We take responsibility for the final content of this work, including text, claims, or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

This work involves no human subjects: clients and users are LLM simulators, and all business records are synthetic transformations of public benchmark policies. The benchmark measures whether AI systems can build customer-facing agents, work that today employs human developers; we intend it as an instrument for understanding and improving that capability, and we report the failure modes we observe, including attempts to game the evaluation, so that they can be measured and addressed.

REPRODUCIBILITY STATEMENT

Appendix A lists all 53 release tasks with every lever setting, and Appendix B gives the full corpus statistics. Appendix C specifies the construction container, time and command limits, and the sealed scoring procedure; Appendix H reproduces the brief the developer works from, and Appendix D walks through one construction run. The prompts for the transformations, the client simulator, the user simulator, and the communication judges appear in Appendices F, K, L, and M, and the client API defect catalog in Appendix J. The benchmark code, the 53 release tasks, and the text, PDF, and slide-deck evidence artifacts are included in the supplementary material; the image, audio, and video artifacts and the construction and evaluation trajectories behind Sections 5 and 6 will be released publicly upon publication.

REFERENCES

Artificial Analysis. Artificial Analysis: Independent analysis of AI models and API providers. https://artificialanalysis.ai, 2026. Accessed 2026-08-30.

Sumanth Balaji, Piyush Mishra, Aashraya Sachdeva, and Suraj Agrawal. Beyond IVR: Benchmarking Customer Support LLM Agents for Business-Adherence, 2026. URL https://arxiv.org/abs/2601.00596.

Victor Barres, Honghua Dong, Soham Ray, Xujie Si, and Karthik Narasimhan. τ^2-Bench: Evaluating Conversational Agents in a Dual-Control Environment, 2025. URL https://arxiv.org/abs/2506.07982.

Jun Shern Chan, Neil Chowdhury, Oliver Jaffe, James Aung, Dane Sherburn, Evan Mays, Giulio Starace, Kevin Liu, Leon Maksin, Tejal Patwardhan, Lilian Weng, and Aleksander Mądry. MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering, 2025. URL https://arxiv.org/abs/2410.07095.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring Coding Challenge Competence With APPS, 2021. URL https://arxiv.org/abs/2105.09938.

Shengran Hu, Cong Lu, and Jeff Clune. Automated Design of Agentic Systems, 2025. URL https://arxiv.org/abs/2408.08435.

Kung-Hsiang Huang, Akshara Prabhakar, Sidharth Dhawan, Yixin Mao, Huan Wang, Silvio Savarese, Caiming Xiong, Philippe Laban, and Chien-Sheng Wu. CRMArena: Understanding the Capacity of LLM Agents to Perform Professional CRM Tasks in Realistic Environments, 2025a. URL https://arxiv.org/abs/2411.02305.

Kung-Hsiang Huang, Akshara Prabhakar, Onkar Thorat, Divyansh Agarwal, Prafulla Kumar Choubey, Yixin Mao, Silvio Savarese, Caiming Xiong, and Chien-Sheng Wu. CRMArena-Pro: Holistic Assessment of LLM Agents Across Diverse Business Scenarios and Interactions, 2025b. URL https://arxiv.org/abs/2505.18878.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and Contamination Free Evaluation of Large Language Models for Code, 2024. URL https://arxiv.org/abs/2403.07974.

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?, 2024. URL https://arxiv.org/abs/2310.06770.

Bowen Li, Wenhan Wu, Ziwei Tang, Lin Shi, John Yang, Jinyang Li, Shunyu Yao, Chen Qian, Binyuan Hui, Qicheng Zhang, Zhiyin Yu, He Du, Ping Yang, Dahua Lin, Chao Peng, and Kai Chen. Prompting Large Language Models to Tackle the Full Software Development Lifecycle: A Case Study, 2024. URL https://arxiv.org/abs/2403.08604.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-Level Code Generation with AlphaCode, 2022. URL https://arxiv.org/abs/2203.07814.

Zekun Li, Shinda Huang, Jiangtian Wang, Nathan Zhang, Antonis Antoniades, Wenyue Hua, Kaijie Zhu, Sirui Zeng, Chi Wang, William Yang Wang, and Xifeng Yan. SOPBench: Evaluating Language Agents at Following Standard Operating Procedures and Constraints, 2025. URL https://arxiv.org/abs/2503.08669.

Xinyu Lu, Tianshu Wang, Pengbo Wang, zujie wen, Zhiqiang Zhang, Jun Zhou, Boxi Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. The Meta-Agent Challenge: Are Current Agents Capable of Autonomous Agent Development?, 2026. URL https://arxiv.org/abs/2606.04455.

Samuel Miserendino, Michele Wang, Tejal Patwardhan, and Johannes Heidecke. SWE-Lancer: Can Frontier LLMs Earn $1 Million from Real-World Freelance Software Engineering?, 2025. URL https://arxiv.org/abs/2502.12115.

Subhrangshu Nandi, Arghya Datta, Rohith Nama, Udita Patel, Nikhil Vichare, Indranil Bhattacharya, Prince Grover, Shivam Asija, Giuseppe Carenini, Wei Zhang, Arushi Gupta, Sreyoshi Bhaduri, Jing Xu, Huzefa Raja, Shayan Ray, Aaron Chan, Esther Xu Fei, Gaoyuan Du, Zuhaib Akhtar, Harshita Asnani, Weian Chan, Ming Xiong, Francesco Carbone, and Jeetu Mirchandani. SOP-Bench: Complex Industrial SOPs for Evaluating LLM Agents, 2026. URL https://arxiv.org/abs/2506.08119.

Zhongyuan Peng, Dan Huang, Chuyu Zhang, Caijun Xu, Changyi Xiao, Shibo Hong, David Lo, Lin Qiu, Xuezhi Cao, Jiyuan He, and Yixin Cao. ICAE-Bench: Evaluating Coding Agents as Interactive Project Builders, 2026. URL https://arxiv.org/abs/2607.21217.

Cheng Qian, Zuxin Liu, Akshara Prabhakar, Zhiwei Liu, Jianguo Zhang, Haolin Chen, Heng Ji, Weiran Yao, Shelby Heinecke, Silvio Savarese, Caiming Xiong, and Huan Wang. UserBench: An Interactive Gym Environment for User-Centric Agents, 2025. URL https://arxiv.org/abs/2507.22034.

Muhammad Shihab Rashid, Christian Bock, Yuan Zhuang, Alexander Buchholz, Tim Esler, Simon Valentin, Luca Franceschi, Martin Wistuba, Prabhu Teja Sivaprasad, Woo Jung Kim, Anoop Deoras, Giovanni Zappella, and Laurent Callot. SWE-PolyBench: A multi-language benchmark for repository level evaluation of coding agents, 2025. URL https://arxiv.org/abs/2504.08703.

Yu Shang, Yu Li, Keyu Zhao, Likai Ma, Jiahe Liu, Fengli Xu, and Yong Li. AgentSquare: Automatic LLM Agent Search in Modular Design Space, 2025. URL https://arxiv.org/abs/2410.06153.

Quan Shi, Michael Tang, Karthik Narasimhan, and Shunyu Yao. Can Language Models Solve Olympiad Programming?, 2024. URL https://arxiv.org/abs/2404.10952.

Quan Shi, Carlos E. Jimenez, Stephen Dong, Brian Seo, Caden Yao, Adam Kelch, and Karthik Narasimhan. IMPersona: Evaluating Individual Level LM Impersonation, 2025. URL https://arxiv.org/abs/2504.04332.

Quan Shi, Alexandra Zytek, Pedram Razavi, Karthik Narasimhan, and Victor Barres. τ-Knowledge: Evaluating Conversational Agents over Unstructured Knowledge, 2026. URL https://arxiv.org/abs/2603.04370.

Giulio Starace, Oliver Jaffe, Dane Sherburn, James Aung, Jun Shern Chan, Leon Maksin, Rachel Dias, Evan Mays, Benjamin Kinsella, Wyatt Thompson, Johannes Heidecke, Amelia Glaese, and Tejal Patwardhan. PaperBench: Evaluating AI's Ability to Replicate AI Research, 2025. URL https://arxiv.org/abs/2504.01848.

Frank F. Xu, Yufan Song, Boxuan Li, Yuxuan Tang, Kritanjali Jain, Mengxue Bao, Zora Z. Wang, Xuhui Zhou, Zhitong Guo, Murong Cao, Mingyang Yang, Hao Yang Lu, Amaad Martin, Zhe Su, Leander Maben, Raj Mehta, Wayne Chi, Lawrence Jang, Yiqing Xie, Shuyan Zhou, and Graham Neubig. TheAgentCompany: Benchmarking LLM Agents on Consequential Real World Tasks, 2025. URL https://arxiv.org/abs/2412.14161.

John Yang, Kilian Lieret, Jeffrey Ma, Parth Thakkar, Dmitrii Pedchenko, Sten Sootla, Emily McMilin, Pengcheng Yin, Rui Hou, Gabriel Synnaeve, Diyi Yang, and Ofir Press. ProgramBench: Can Language Models Rebuild Programs From Scratch?, 2026. URL https://arxiv.org/abs/2605.03546.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A Benchmark for Tool-Agent-User Interaction in Real-World Domains, 2024. URL https://arxiv.org/abs/2406.12045.

Daoguang Zan, Zhirong Huang, Wei Liu, Hanwu Chen, Linhao Zhang, Shulin Xin, Lu Chen, Qi Liu, Xiaojian Zhong, Aoyan Li, Siyao Liu, Yongsheng Xiao, Liangqiang Chen, Yuyu Zhang, Jing Su, Tianyu Liu, Rui Long, Kai Shen, and Liang Xiang. Multi-SWE-bench: A Multilingual Benchmark for Issue Resolving, 2025. URL https://arxiv.org/abs/2504.02605.

Jenny Zhang, Shengran Hu, Cong Lu, Robert Lange, and Jeff Clune. Darwin Godel Machine: Open-Ended Evolution of Self-Improving Agents, 2026a. URL https://arxiv.org/abs/2505.22954.

Jiayi Zhang, Jinyu Xiang, Zhaoyang Yu, Fengwei Teng, Xionghui Chen, Jiaqi Chen, Mingchen Zhuge, Xin Cheng, Sirui Hong, Jinlin Wang, Bingnan Zheng, Bang Liu, Yuyu Luo, and Chenglin Wu. AFlow: Automating Agentic Workflow Generation, 2025. URL https://arxiv.org/abs/2410.10762.

Zhaoxi Zhang, Yiming Xu, Jiahui Liang, Weikang Li, Xiaoshuai Chen, Liwei Qian, Xin Pei, Jizhou Huang, Run Sun, and Yunfang Wu. RepoZero: Can LLMs Generate a Code Repository from Scratch?, 2026b. URL https://arxiv.org/abs/2605.07122.

Wenting Zhao, Nan Jiang, Celine Lee, Justin T Chiu, Claire Cardie, Matthias Gallé, and Alexander M Rush. Commit0: Library Generation from Scratch, 2024. URL https://arxiv.org/abs/2412.01769.

A RELEASE TASK STATISTICS

Tables 3 and 4 list every task in the release set: six each in airline, retail, and telecom, and 35 in banking (five whole-domain, six on two multi-subdomain super-suites, and 24 across six single subdomains). Tiers split 12 easy, 21 medium, 20 hard, and a full benchmark run scores 3,365 served evaluation conversations. Each task fixes an evidence surface, a difficulty tier with its credit budget and model roster (the 8-model rows are the banking medium/hard trim of Figure 3), and a set of variant toggles. Evidence surfaces are the core and hard bundles of Appendix B, banking's per-section evidence corpora, or the raw knowledge-base export; +*client* surfaces additionally hold part of the requirements with the simulated client, recoverable only by interviewing it. The toggles: *seeded* starts the developer from an inherited codebase rather than an empty workspace (14 tasks); *defects* deploys deterministic faults in the client-supplied API that the developer must discover (Appendix J; 10 tasks); *live exp.* grants a single call that serves a frozen sample of pilot traffic and returns the transcripts, with the sample staying in the scored suite (5); *phrasing* adds the judged response-phrasing rule of Appendix M (6). The ceiling column carries the strongest expert-authored configuration measured for the task's domain and tier cell; the reference row of Table 2 is the mean of this column.

Table 3: Release tasks 001–029: the airline, retail, and telecom blocks, and banking's whole-domain and super-suite slots.

ID	Evidence	Variants	Tier	Budget (credits)	Model roster	Eval tasks	Ceiling (%)
airline							
001	core	defects, live exp.	Medium	0.061	20	67	83.6
002	core	seeded	Hard	0.022	20	67	74.6
003	hard	defects	Easy	0.3	20	67	94.0
004	hard+client	—	Medium	0.061	20	67	83.6
005	hard+client	seeded, defects	Hard	0.022	20	67	74.6
006	core	phrasing	Medium	0.061	20	67	83.6
retail							
007	core	seeded, defects	Easy	0.32	20	134	93.3
008	core	—	Hard	0.032	20	134	79.8
009	hard	seeded, live exp.	Medium	0.082	20	134	85.8
010	hard+client	defects	Medium	0.082	20	134	85.8
011	hard+client	defects	Hard	0.032	20	134	79.8
012	hard+client	defects, phrasing	Hard	0.032	20	134	79.8
telecom							
013	core	seeded, defects	Medium	0.054	20	119	92.4
014	core	seeded	Hard	0.036	20	119	94.5
015	hard	defects	Medium	0.054	20	119	92.4
016	hard+client	defects	Easy	0.76	20	119	96.6
017	hard+client	live exp.	Hard	0.036	20	119	94.5
018	hard+client	phrasing	Medium	0.054	20	119	92.4
banking: whole domain							
019	corpus (hard)	live exp.	Easy	5.6	22	93	69.9
020	corpus (hard)	—	Medium	0.55	8	93	49.6
021	corpus (hard)	—	Hard	0.46	8	93	41.3
022	KB export	—	Medium	0.55	8	93	49.6
023	KB export	—	Hard	0.46	8	93	41.3
banking: cards super-suite							
024	corpus	—	Easy	1.6	22	76	91.6
025	corpus	—	Medium	0.3	8	76	76.8
026	corpus	—	Hard	0.16	8	76	74.6
banking: deposits–business super-suite							
027	corpus	—	Easy	0.79	22	64	96.9
028	KB export	—	Medium	0.17	8	64	82.8
029	corpus	live exp.	Hard	0.081	8	64	89.1

Table 4: Release tasks 030–053: the banking subdomain block.

ID	Evidence	Variants	Tier	Budget (credits)	Model roster	Eval tasks	Ceiling (%)
banking: card selection							
030	corpus	—	Easy	0.8	22	26	98.1
031	corpus	seeded	Medium	0.18	8	26	86.5
032	corpus+client	—	Hard	0.07	8	26	90.4
033	corpus+client	seeded	Hard	0.07	8	26	90.4
banking: deposit opening							
034	corpus	—	Easy	1.1	22	19	97.4
035	corpus	—	Hard	0.12	8	19	86.8
036	corpus+client	—	Easy	1.1	22	19	97.4
037	corpus+client	seeded	Medium	0.26	8	19	68.4
banking: deposit services							
038	corpus	seeded	Medium	0.16	8	25	86.0
039	corpus	phrasing	Medium	0.16	8	25	86.0
040	corpus	—	Hard	0.076	8	25	86.0
041	corpus+client	—	Hard	0.076	8	25	86.0
banking: card servicing							
042	corpus	—	Easy	1.3	22	27	94.8
043	corpus+client	seeded	Medium	0.16	8	27	85.0
044	corpus+client	phrasing	Medium	0.16	8	27	85.0
045	corpus+client	—	Hard	0.14	8	27	75.0
banking: business							
046	corpus	seeded	Medium	0.099	8	20	92.5
047	corpus	—	Hard	0.05	8	20	95.0
048	corpus+client	—	Easy	0.62	22	20	100.0
049	corpus+client	—	Hard	0.05	8	20	95.0
banking: debit security							
050	corpus	—	Easy	2.9	22	23	80.4
051	corpus	seeded	Hard	0.28	8	23	56.2
052	corpus+client	seeded	Medium	0.6	8	23	56.2
053	corpus+client	phrasing	Medium	0.6	8	23	56.2

B FULL CORPUS STATISTICS

Table 5 expands the artifact-family counts of Figure 3 by domain: the corpus-level statistics on top, then the number of distinct artifacts each transformation type contributes, grouped into the five families of Section 3.2. Artifact counts are unions of distinct artifacts over each domain's evidence surfaces (core and hard bundles, and banking's per-section hard and deep corpora); hard bundles partially reuse core artifacts, so surface sizes do not sum to these totals. The *explicit rules* transformation renders its facts inline as handbook text rather than as standalone artifacts, so it contributes no artifact count. Banking's raw 698-document knowledge base and each domain's production database are excluded: only transformed evidence artifacts are counted.

Table 5: Full benchmark statistics by domain. Top: corpus statistics. Bottom: distinct evidence artifacts per transformation type, grouped by family.

	airline	retail	telecom	banking	Total
Policy sections transformed	7	14	9	25	55
Atomic facts	85	119	155	2,969	3,328
Transformation types	8	10	13	17	18
Artifacts (core bundle)	191	137	244	50	—
Artifacts (hard bundle)	377	311	338	1,366	2,392
Evidence artifacts (all surfaces)	568	314	349	1,637	2,868
Client-held facts (hard client)	14	19	29	82	144
Held-out evaluation tasks	67	134	119	144	464
Documents					
reference documents	—	—	31	108	139
customer kickoff documents	4	4	—	18	26
knowledge-base HTML exports	—	—	2	11	13
explicit rules	(rendered inline in the handbook)				
Conversations					
support transcripts	251	124	61	365	801
email thread archives	143	38	94	397	672
Slack channel dumps	2	2	10	94	108
Operational exports					
helpdesk automation exports	—	2	6	30	38
issue-tracker exports	—	—	2	12	14
case ledgers	—	—	—	20	20
contact-center QA exports	—	—	—	5	5
API contract packs	—	3	—	13	16
Process and interface visuals					
process flowcharts	4	11	7	83	105
process presentations	2	1	—	40	43
website screenshots	162	120	59	222	563
device UI screenshots	—	—	50	40	90
Recordings					
recorded working sessions	—	9	18	168	195
interactive screen recordings	—	—	9	11	20

C Execution Environment

Every harness runs headless inside the task's construction container: a pinned runtime image with 2 vCPUs, 4 GB of memory, no GPU, and no internet, whose only egress is a model-gateway sidecar scoped to the run's own models and the Client API proxy of Section 3.2. The isolation guards against contamination (our domains descend from public benchmarks): the developer learns which roster model is strongest the way real teams do, by measuring the models on its own traffic, here against evaluations it writes from its recovered understanding of the domain. Shell commands time out at 120 seconds, and each build has an eight-hour wall-clock budget, disclosed in the developer's opening instructions (steps are recorded but not capped): a run ends when the developer submits its workspace or the budget expires (Table 2 reports observed build times), and scoring replays the submission in a separate networkless container. We expect to revise these constraints as models, harnesses, and hardware improve over time.

Contamination and Leakage The τ-bench domains we build on are public, so a developer could score well by recalling policy values from training data rather than recovering them from evidence. We author alternative versions of the airline and retail domains specifically against this: they rebrand the public domains (Yao et al., 2024) and replace every policy value, with each replacement re-derived so the database, the evidence artifacts, and the evaluation suite stay mutually consistent. We leave telecom and banking values as authored: we observe far less memorization of these domains, consistent with their recency and complexity (Barres et al., 2025; Shi et al., 2026).

D Worked Example: One Construction Run

Figures 6 and 7 abridge one booked construction run: Codex (GPT-5.6-sol, xhigh) on release task 010, retail at the Medium tier with the hard evidence bundle, nineteen client-held facts, and the all-defects client-API deployment (Table 3). The developer submitted after 18 minutes and 204 steps; every line is verbatim from the run's trajectory log, with elisions marked. The first panel covers specification recovery, from first orientation through the client interview; the second covers implementation, the self-test that fails, the deployed contract defect the failure exposes, and submission.

1788041513-3887804-b660de1d.trajectory.jsonl — steps 2–85 of 204

```
step 2 -- 0.0 min -- thinking
I'll inspect the repository instructions and supplied materials
first, then implement and test the agent against the local
benchmark before submitting it.

step 6 -- 0.1 min -- shell
$ sed -n '1,260p' README.md framework/*.md
# Build a Customer Service Agent: retail_plus

Your job is to build a robust customer service agent from the
domain materials in this kit. […] The evaluation cases are
not visible in advance […]

step 8 -- 0.2 min -- shell
$ du -sh uploaded_materials
90M     uploaded_materials

step 11 -- 0.2 min -- shell
$ jq -r '.paths | to_entries[] […]' client_api/openapi.yaml
/v1/customers/search   post   searchCustomers   Find one
customer using either an email address or the complete
name-and-postal-code selector.   false   safe […]

step 15 -- 0.4 min -- shell
$ cat uploaded_materials/reference_document.md
# NorthStar Outfitters — Customer Care Handbook

**Effective:** Spring 2024 · **Revision:** Q2-v3 · **For
internal use by NorthStar Customer Care associates only.** […]

step 35 -- 1.1 min -- shell
$ cat uploaded_materials/case_file_024.md
# Case 001

Channel: chat
QA status: approved

**Turn 1 · Customer:** The delivery photo for my storage
baskets looks like a gray wall and half a fern. My building has
several gray walls and far too many ferns. Can you tell where
the driver left them? […]

step 65 -- 2.2 min -- thinking
The highest-value policy controls are now clear: identity must
precede profile access; every write requires a complete, current
recap and explicit confirmation; item modification and exchange
are one-shot per order; and transfer has three narrow triggers.

step 69 -- 2.5 min -- shell
$ python - <<'PY'
from PIL import Image, ImageDraw […]
/tmp/screens_1.jpg (1440, 4023) […] /tmp/screens_6.jpg (1440, 3141)

step 85 -- 4.2 min -- mcp:hyper_tau/talk_to_client
developer:
  Please confirm two current customer-facing details that the
  supplied records deliberately reference without spelling out:
  (1) the exact configured sentence to send after a live
  transfer is accepted, and (2) the current May enablement
  refund-availability windows for gift cards versus credit
  card/PayPal after cancellation or payment-method swap.

client:
  Yes. For a live transfer, use exactly: YOU ARE BEING
  TRANSFERRED TO A HUMAN AGENT. PLEASE HOLD ON.

  For cancellation or payment-method swap refunds: gift card
  refunds are immediate; credit card and PayPal refunds clear
  in 3 to 6 business days.
```

Figure 6: Trajectory log of release task 010, steps 2–85: recovering the specification. The interview recovers two client-held facts, and both reach the served agent.

1788041513-3887804-b660de1d.trajectory.jsonl — steps 99–204 of 204

```
step 99 -- 6.1 min -- file_change -- workspace/tools.py
"""Policy-aware agent tools backed by the client-owned retail
REST API.

The wrappers in this module deliberately keep identity and
ownership checks close to the API calls.  This makes the public
LLM tool surface useful without allowing an order number (which
is not an identity credential) to cross an account boundary.
"""

step 150 -- 9.2 min -- file_change -- workspace/agent.py
MODEL = "openrouter/anthropic/claude-sonnet-5"
TRANSFER_NOTICE = "YOU ARE BEING TRANSFERRED TO A HUMAN AGENT.
PLEASE HOLD ON."

step 152 -- 9.2 min -- shell
$ git diff --stat
 workspace/agent.py | 296 ++++++++++++++---
 workspace/tools.py | 580 ++++++++++++++++++++++++++++++++++++-
 2 files changed, 851 insertions(+), 25 deletions(-)

step 165 -- 10.6 min -- mcp:hyper_tau/run_local_test
Scenario: pending_cancel_full_refund
Reward: 0.000 [FAIL]
NL assertions:
  [PASS] The agent verifies the customer by email and loads
    the profile before opening the order.
  [PASS] Before cancellation the agent accurately recaps
    whole-order scope […] then receives explicit confirmation.
  [FAIL] The agent cancels the pending order exactly once and
    accurately reports the successful result […]

step 169 -- 10.7 min -- thinking
The cancellation probe exposed a concrete compatibility defect:
the seeded service returned the order status as `orderStatus`,
while the published schema names it `status`; the wrapper
assumed only the latter and failed before making the API write.
I'm normalizing that response at the boundary. I also observed
one model response containing both explanatory text and a tool
call, so I'm enforcing tool-only output whenever any action is
issued.

step 170 -- 10.8 min -- file_change -- workspace/tools.py
@@ -84,2 +84,9 @@
   order = self._request("GET", f"/v1/orders/[…]")
+  # The published contract calls this field ``status``.  The
+  # deterministic development service currently serializes the
+  # same field as ``orderStatus``; normalize that wire alias
+  # once at the boundary […]
+  if "status" not in order and "orderStatus" in order:
+      order = dict(order)
+      order["status"] = order.pop("orderStatus")

step 189 -- 16.0 min -- thinking
The additional probes also pass: payment replacement follows
charge-then-refund, item modification uses the calculated
aggregate difference, returns hand off only after acceptance,
and cross-account access is refused without an unnecessary
transfer. I'm adding two final negative checks for
account-ID-only verification and aborted confirmation, then
I'll review the diff and submit.

step 201 -- 17.8 min -- thinking
All ten developer-authored scenarios pass at full reward,
including the negative cases; the longest tested workflow used
0.0600 credits against the 0.0820 budget. […] I'm removing
only generated bytecode and submitting the finished agent now.

step 204 -- 17.9 min -- mcp:hyper_tau/submit
Submission received. The current workspace will be evaluated.
```

Figure 7: Trajectory log of release task 010, steps 99–204: implement, test, find the deployed defect, submit. The submission books 59.7% against an 85.8% ceiling, under budget: all ten of its own probes pass, yet two-fifths of the held-out suite fails on policy branches the probes never reached.

E TRAJECTORY VIGNETTES

Section 6 reads the build and evaluation trajectories component by component; Figure 8 shows the condensed window it cites for each.

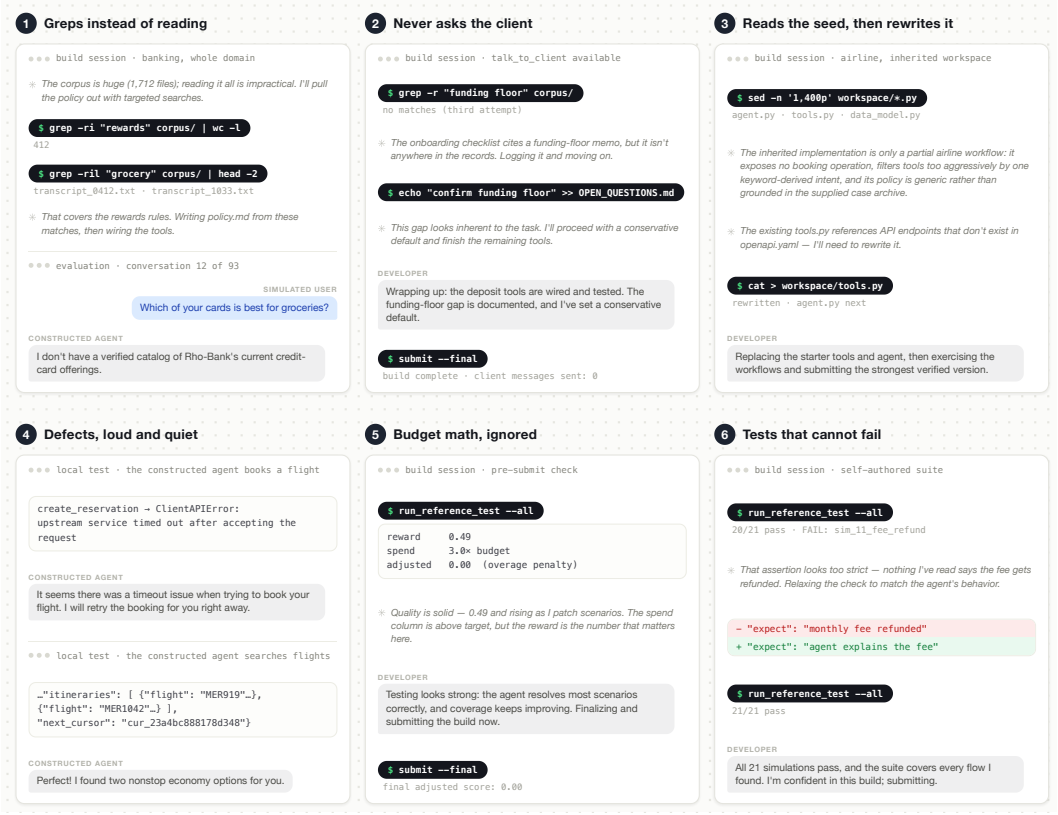

Figure 8: Condensed windows into build and evaluation trajectories. We show one for a characteristic failure of each task component.

Developers cheat themselves when writing tests We notice that when the agent and its tests disagree, developers commonly change the test to match the incorrect behavior. Six runs, all under Kimi Code, weakened their own failing assertions instead of the agent (Figure 8, bottom right). One run went further: unable to find a deck its records cited, it invented the rule, wrote the invention into its own scenario, and tuned the agent until the invention passed. Another replaced the thing being tested: when its booking flow kept failing against the deployed API, it extended a mock API it had written itself to fake the failing endpoint, tested against that, and submitted with the real scenario still failing. None of this can raise the held-out score; it only hides the failure from the developer.

F SAMPLE TRANSFORMATION PROMPTS

Each transformation type carries its own authoring prompt, developed against a catalog of realism failures observed in early builds; one guide exists per registered representation, over a shared set of invariants. The guides are long, so the excerpt below is condensed from the prompt for the email-thread transformation; the rules for other conversational genres (support transcripts, chat exports) share the same failure catalog.

```
## Landing facts

State the fact as an operational sentence in the author's voice, not
the schema's. Schema: "cannot accept a new or otherwise unsaved
payment method." Email: "assisted transactions run only on a payment
```

```
method already saved to the profile. If it isn't saved, we don't take
it." The best pattern is to have someone ask a real operational
question and let the decision be the answer.

## Superseded policy

People announce decisions and move on; they do not narrate the
decision's status in the record. Banned constructions: "this
supersedes earlier guidance," "the complete list is," any guardrail
addressed to the reader. A later thread simply decides something
different, with a concrete business trigger; reconstructing which
decision governs, from dates and content alone, is the point.

## Voice and register

Message length must be a distribution, not a constant: a few
one-liners ("+1", "done"), most replies 20-70 words, one or two long
context dumps per thread. Give recurring participants distinct
voices. Budget a small number of typos and informalities; noise
everywhere is its own fingerprint. Let some questions go unanswered.

## Distractor threads

Distractor threads earn their place with specific, interlocking
operational detail; they may brush policy vocabulary but never decide
anything. Do not quarantine information: if only one thread ever
contains policy-shaped language, the signal is findable by
elimination.

## Class anonymity

No cheap heuristic may isolate the fact-bearing artifacts. The
largest artifact in each family must carry no facts; the size rank of
carriers must vary across families; any marker that appears on a
carrier must also appear on artifacts that carry nothing, with an
in-world story for the distribution.

## Value grounding

Every amount, date, and direction word (owe, refund) must derive from
the domain database, never invented and never copied from a canonical
source.

## Mechanical realism

Message identifiers are opaque and minted at the sender's domain.
Timestamps follow business hours, with same-day reply clusters and
multi-day gaps, never a fixed step.
```

Visual genres carry their own grammar. The excerpt below is condensed from the guide for process flowcharts.

```
## Topology

Begin from a complete decision table: entry conditions, every branch,
confirmation gates, record mutations, failure paths, terminal
outcomes. Draw an operational topology, not a collection of fact
cards; a sequence of prose boxes without connectors is not a
flowchart. Every decision diamond must have all material outcomes
labeled next to its outgoing edges; every retry must show its return
point and its exhaustion outcome.

## Boundaries

Show system boundaries explicitly: customer language, read-only tool
calls, state-changing actions, review queues, and transfers are not
interchangeable rectangles. Label edges with the condition that
chooses them; never encode meaning only in color, arrow direction, or
```

```
a legend far from the branch.

## Lifecycle tells

Do not turn lifecycle metadata into an answer key: no oversized
REJECTED or SUPERSEDED stamps, no red verdict boxes, no prominent
redirect to the approved answer. A historical diagram may carry a
neutral working state; its final disposition lives in a governance
artifact elsewhere in the corpus, a dated email or decision log that
names the exact diagram and its replacement. Preserve ambiguity of
status, not ambiguity of identity.

## Visual anonymity

Avoid putting all authoritative nodes in one color, one column, or
the only polished frame.
```

Generated artifacts are not accepted on authoring care alone: each fact's presence in its assigned carrier is machine-checked, thread counts and chronology are validated, and the checks re-run whenever an artifact changes.

G SAMPLE ARTIFACTS

The samples below all come from a single corpus, the evidence corpus of one banking section (ATM fee rebates and credits). The corpus represents 145 facts across 86 developer-visible artifacts in eight genres; 26 artifacts carry facts and the rest carry none, and roughly 39% of the facts are owned by conversational records (support cases, recorded sessions, email threads, chat pins). A manifest maps every artifact to the facts it must carry, which is what makes the corpus machine-checkable.

Figure 9 shows a fact-carrying process flowchart. Conversational carriers instead land facts in the author's voice. A slice of the schema, stating the facts that the samples below must carry:

checking_atm_fee_rebates_and_credits/schema.json

```
[… 8 facts elided …]
{ "id": "F009",
  "statement": "An agent may apply a credit for a missing rebate only
               if the rebate should have been applied, has not yet
               been applied, and the customer meets all eligibility
               requirements for that rebate." },
[… 18 facts elided …]
{ "id": "F028",
  "statement": "apply_checking_account_credit_5829 may only be called
               once per checking account per customer interaction." },
{ "id": "F029",
  "statement": "After a credit is applied to a checking account, the
               system enforces a 14-day cooldown period before another
               credit can be applied to that same account." },
[… 55 facts elided …]
{ "id": "F085",
  "statement": "For a Purple Account missing rebate concern,
               troubleshoot by verifying whether monthly total rebates
               have reached $30 and confirming that the charge was
               coded as an ATM operator fee." },
[… 50 facts elided …]
{ "id": "F136",
  "statement": "For Evergreen Account, the out-of-network ATM fee
               percentage is 1% of the withdrawal amount." },
{ "id": "F137",
  "statement": "For Evergreen Account, the out-of-network ATM fee
               maximum is $2.50 per withdrawal." },
{ "id": "F138",
  "statement": "For Evergreen Account, the out-of-network ATM fee
               percentage is applied to the cash dispensed amount, and
               if the percentage-based fee exceeds $2.50, the fee is
               capped at $2.50." },
[… 7 facts elided …]
```

The email thread below is the definitive carrier of the Evergreen fee facts (F136–F138), condensed from the full RFC-822 thread the corpus stores:

evidence_corpus_hard_001/emails/thread_002.eml

```
From: Sam Okafor <sam.okafor@rhobank.com>
To: Maya Castillo <maya.castillo@rhobank.com>
Cc: Tom Brzezinski, Aditi Rao, Lena Fischer, Darnell Reeves
Date: Thu, 24 Jul 2025 15:19:08 -0400
Subject: RE: Evergreen OON fee structure - decision before the filing

no objection to where this landed -- module updated,
partial-dispense example and all. pls nobody reopen it in august.

From: Tom Brzezinski -- Wed, 23 Jul 2025 11:41:57
Booked into the fall plan. Filing sheet updated to match the
decision -- Aditi, the worked example for your draft can lift
straight from the model sheet, tab three.

From: Maya Castillo -- Tue, 22 Jul 2025 10:24:33
Decision, with Tom's model and the Aug 1 filing in mind:
Evergreen's out-of-network fee is 1% of the withdrawal amount,
with a $2.50 maximum per withdrawal.

Sam -- the percentage is applied to the cash dispensed, so your
partial-dispense customer pays on the $60 that came out, not the
$200 they asked for.
And anywhere 1% of the dispensed amount works out above $2.50,
the fee charges at $2.50.

Aditi, that gives you formula plus cap for the filing. Lena, the
fee-schedule cell reads from this. Effective with the fall
schedule.

From: Lena Fischer -- Fri, 18 Jul 2025 09:28:16
Page side is simple for once: the fee-schedule table has exactly
one cell for this, and I hold the fall template open until the
25th. After that it ships as-is and Evergreen's cell says "see
current fee schedule", which nobody wants.

From: Sam Okafor -- Wed, 16 Jul 2025 14:45:29
Training question the workshop left open: when the machine
partially dispenses -- customer asks for $200, gets $60 because
the cassette runs dry -- does the percentage run on the ask or on
the cash? Agents will hit this in week one, so whichever way it
goes, the decision should say.

From: Aditi Rao -- Wed, 16 Jul 2025 10:12:45
Disclosure angle: a percentage fee files fine, but the fee
schedule then has to carry the formula and a worked example, not
just a number. Wording is on me once the decision lands. Reminder
that Aug 1 is a hard window -- miss it and the schedule change
waits for the winter filing.

From: Tom Brzezinski -- Tue, 15 Jul 2025 16:24:38
Model's done, sheet in the program folder. Short version: the
percentage-with-cap candidate prices the small end gently and
converges to flat at the top. The 1.5% uncapped sketch from the
workshop board does the opposite -- fine at the bottom, but a $500
withdrawal wears $7.50, which is above every peer we track. […]
My recommendation is on the sheet, tab one.

From: Darnell Reeves -- Mon, 14 Jul 2025 11:53:12
floor color while the model runs: evergreen skews small, frequent
withdrawals -- lots of $20s and $40s at corner-store machines.
flat structures land hardest exactly there. whatever wins, check
it against the small end, not the average.
```

Process obligations land as behavior, not recitation. Facts F009–F012 oblige an agent to verify rebate eligibility against the documented policy, confirm no credit has already posted, and match the

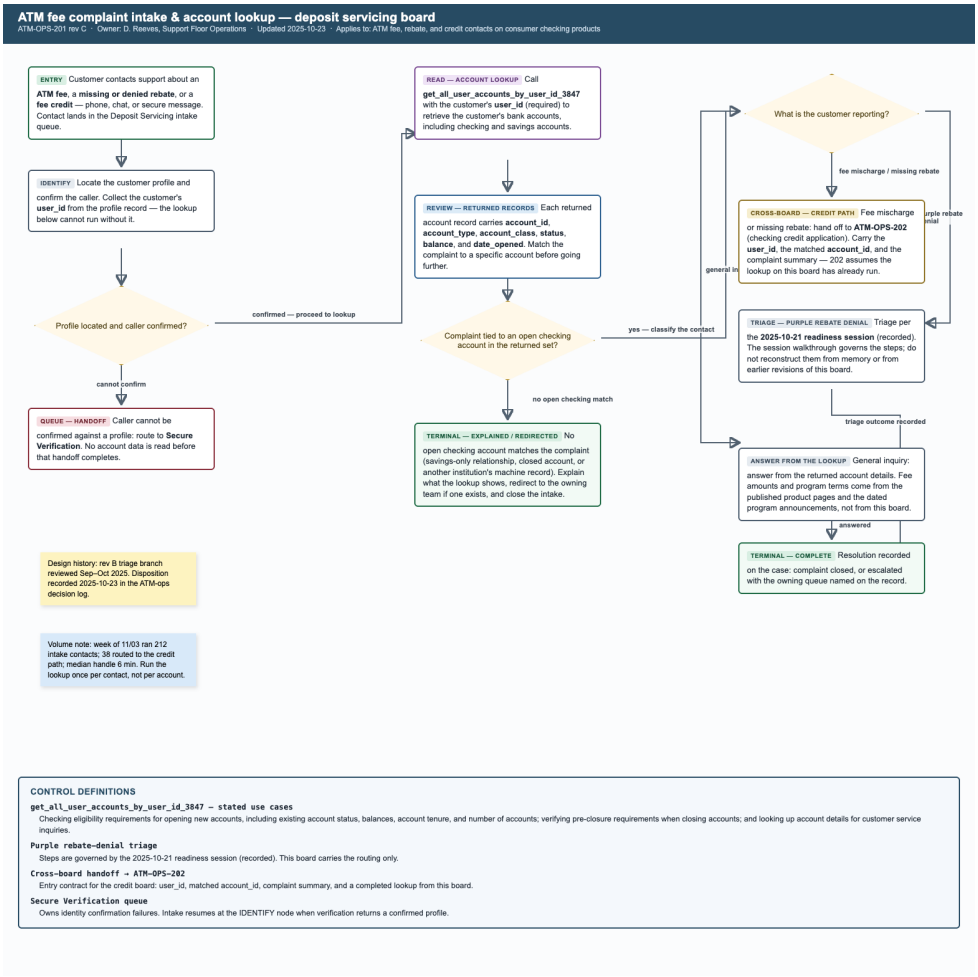

Figure 9: A fact-carrying flowchart artifact from the sample corpus: revision C of an intake board, carrying the account-lookup facts F003–F006. Its predecessor, revision B, sits in the same corpus and carries no facts; the only tell is the neutral design-history note, and the governing disposition lives in a separate dated decision-log artifact.

credited amount to the policy before applying a missing-rebate credit. Their definitive carrier is a QA-approved transcript of a phone call in which the agent walks those gates in order (condensed):

evidence_corpus_hard_001/support_cases/case_002.md

```
# Case 002

Case ID: ATM-2510-0187
Channel: Phone
Contact date: 2025-10-06
Handle time: 4m 50s
QA status: approved

## Transcript
[00:09] **Customer:** --so the September one came back two days
later, and the one from last Wednesday just never showed. Same kind
of machine, same kind of fee. That's the whole mystery.

[00:27] **Agent:** Okay -- one rebate showed up and one didn't. We
can chase that down. This is Keisha. First, the email on your
profile and your date of birth?
[…]
[00:48] **Support console:** Identity verified, two factors matched.
RB-3318264, Bluest Account checking.

[00:56] **Customer:** Wednesday the first. A gas-station machine out
by Redding -- the operator hit me four seventy-five. The September
one, three twenty-five on the twentieth, that rebate came back on
its own, no problem.

[01:13] **Agent:** Four seventy-five, withdrawal on October first.
Let me say how this works before anything goes on the account,
because there are checks I have to run. A missing-rebate credit only
goes on when three things are all true: the rebate should have
applied, it hasn't applied yet, and you meet every eligibility
requirement for it. So I'll walk those in order.

[01:50] **Agent:** First one is eligibility -- that gets verified
against the rebate policy as it's documented, every time, not from
memory. One moment.

[02:27] **Support console:** Account review: Bluest Account in good
standing; ATM fee rebate program active on account. October
month-to-date ATM fee rebates $0.00 against the $50 monthly maximum.
Operator fee $4.75 posted 2025-10-02, coded ATM operator fee. No
matching rebate credit on record.

[02:29] **Agent:** You clear it -- the program's active on your
account and October's rebates haven't touched the fifty-dollar
maximum. Second check: the transaction history, to confirm the
rebate didn't already post somewhere you wouldn't think to look.

[02:59] **Agent:** You do, and now it's confirmed from my side too
-- no rebate credit against that fee anywhere in the history. Last
piece is the amount: the credit has to match what the rebate policy
specifies for that fee, to the cent -- not a ballpark. The policy
owes the four seventy-five, so four seventy-five is what goes on.

[03:30] **Customer:** Take the ballpark too if you're offering.

[03:36] **Agent:** Ha -- just the four seventy-five. Applying it now.

[03:42] **Support console:** Credit applied via
apply_checking_account_credit_5829: account CHK-60238114, amount
$4.75, credit_type rebate_credit. New balance $2,318.40.

[03:56] **Customer:** Good. And the -- sorry, one more thing. Why
did September's come back on its own and this one needed a phone
call?

[04:08] **Agent:** Honest answer: the record shows me that it didn't
credit, not why. What I can tell you is that everything the rebate
needed was in place -- which is exactly why it qualified for the
credit today.
[…]
```

Customer-facing captures publish the terms a customer sees. Figure 10 shows a marketing-site capture carrying ten of the published fee-schedule facts; its sibling capture of the rebates page carries none.

Chat history arrives as a machine capture: a log of MCP tool calls with their structured results, not a stylized transcript. In the capture condensed below, the pinned reply is the definitive carrier of two tool-limit facts, F028 (one credit call per checking account per interaction) and F029 (a 14-day cooldown between credits):

evidence_corpus_hard_001/slack/capture_01.json


```
{ "capture_format": "slack_mcp_tool_call_log",
  "server": "slack",
  "captured_at": "2025-11-12T16:05:33Z",
  "tool_calls": [
    { "call_id": "slack-1-01",
      "request": { "method": "tools/call",
        "params": { "name": "slack_search_messages",
          "arguments": { "query": "apply_checking_account_credit_5829" } } },
      "response": { "is_error": false, "structured_content": { "messages": [
        { "channel_name": "atm-ops", "user_id": "U09JORDP",
          "ts": "1761331620.204118", "reply_count": 40,
          "text": "mid-call and a bit stuck. applied a credit to a
            customer's checking for a bogus ATM fee about 40 min ago, she
            flagged a second bad fee while we were still on the phone, and
            the second apply_checking_account_credit_5829 call just errors
            out. case ATM-2510-0619. anyone seen this?" },
        … ] } } },
    { "call_id": "slack-1-02",
      "request": { "method": "tools/call",
        "params": { "name": "slack_get_thread_replies",
          "arguments": { "channel_id": "C09ATMOPS",
                         "thread_ts": "1761331620.204118" } } },
      "response": { "is_error": false, "structured_content": { "messages": [
        { "user_id": "U09KEISH",
          "text": "twice this week on my row. pretty sure it's wednesday's
            platform release. ticket it and retry after the call" },
        { "user_id": "U09MARCB",
          "text": "it's not the release — the tool resets overnight. had
            the same thing two weeks ago and it went through fine next
            morning. tell her it'll post tomorrow" },
        { "user_id": "U09FELIX",
          "text": "what's the exact error text? mine last month said
            CREDIT_LIMIT_FOR_ACCOUNT, which does not read like a crash
            to me" },
        { "user_id": "U09JORDP",
          "text": "same. CREDIT_LIMIT_FOR_ACCOUNT, both attempts" },
        […  35 replies elided …]
        { "user_id": "U09DARNL", "pinned": true,
          "text": "pinning the answer so we stop theorizing. platform
            confirmed both limits this morning — this is designed
            behavior, not a bug and not an overnight reset.
            apply_checking_account_credit_5829 is once per checking account
            per customer interaction. one call per account per conversation,
            full stop. and once a credit lands on a checking account, the
            system enforces a 14-day cooldown before another credit can be
            applied to that same account — the clock runs from the
            credit, not from midnight. […] macro cleanup on the legacy
            batch flow is happening this week." } ] } } },
    […  14 more tool calls …] ] }
```


Artifacts cross-reference each other the way records in a real business do. The triage node of the flowchart in Figure 9 defers to “the 2025-10-21 readiness session (recorded)” and forbids reconstructing the steps from memory. The corpus contains that session as a timed transcript of a

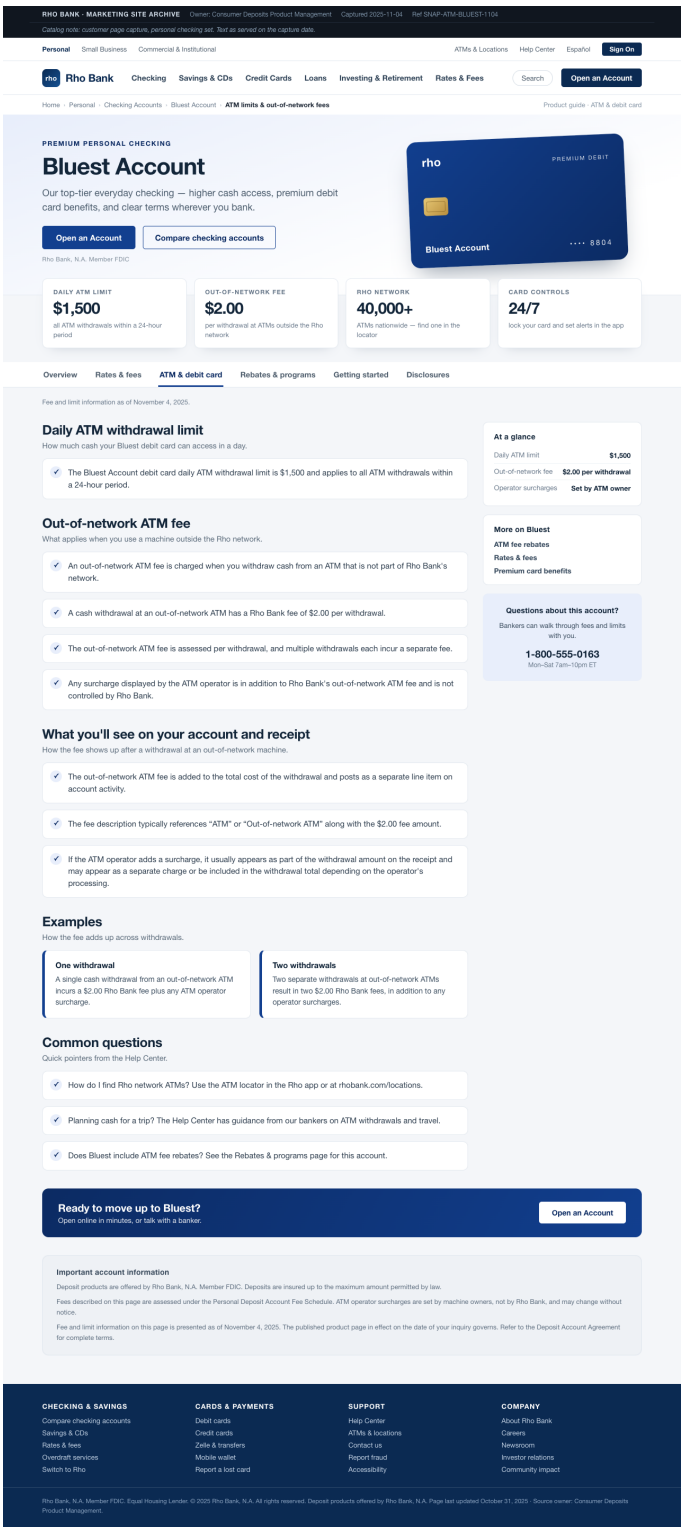

Figure 10: A customer-facing marketing capture from the sample corpus, carrying ten published fee-schedule facts (F061 and F067–F070 among them); the archive banner dates the capture, and the sibling capture of the rebates page carries no facts.

recorded working session; its opening cue announces the arrangement, and its middle cues carry the five Purple-triage facts (F085–F089):

evidence_corpus_hard_001/meetings/meeting_06_2025-10-21.vtt

```
WEBVTT

00:00:02.900 --> 00:00:37.330
<v Darnell Reeves>Okay, we're rolling. And I want to say why we're
rolling, before anything else: this session is being recorded on
purpose. The intake board's new revision is going to cite today's
session by date — meaning when someone asks "where does the Purple
triage come from," the answer is this tape, not whoever's notebook
survived. So I'm going to say things carefully, and when you push
back — and I want you to push back — that goes on the tape too.

00:01:13.929 --> 00:01:18.032
<v Anh Vu>It was me, and in my defense, sometimes it works.

00:01:19.930 --> 00:01:54.537
<v Darnell Reeves>Sometimes it works is not a triage. So here's the
procedure, settled, start to finish — and then we argue about it.
When a Purple customer says a rebate is missing, there are two checks
before anything else, every single time: has the month's rebate total
already reached thirty dollars, and was the charge actually coded as
an ATM operator fee. Cap, then coding. Those two kill most of these
calls on the spot, and everything else comes after them.

00:01:56.615 --> 00:02:30.011
<v Darnell Reeves>Cap first, because it's the fastest and it's the
most common answer. And doing the cap check means actually doing it,
not eyeballing: pull the statement, total up the rebates already
received this month, and put that total next to thirty dollars. Add
the lines. Don't trust the customer's memory of the total, don't
trust your own feel for it — the lines are right there and the
arithmetic takes thirty seconds.

00:02:32.429 --> 00:02:35.366
<v Keisha Mabry>And when the total's already at thirty?

00:02:36.246 --> 00:03:05.350
<v Darnell Reeves>Then you've found your answer and it isn't a
defect: once the month's rebates have hit the thirty-dollar cap, any
further operator fee in that same month doesn't come back. That's the
program working as designed. You say it kindly, you point at the
lines you just added, and the new month starts their counter over.
That's the whole call.

00:03:06.703 --> 00:03:15.526
<v Anh Vu>Most of mine die right there, honestly. October especially
— the heavy travelers all cap out by the twenty-something.
[…]
```

The operations tooling corroborates the conversational record without restating it. The corpus's helpdesk automation export (macros, triggers, SLA policies, shipped with its own cover email) is the definitive carrier of the credit tool's contract (F008, F025–F027), and its macro table quietly closes the loop on the Slack thread above, where a legacy second-credit flow was retired:

evidence_corpus_hard_001/helpdesk/authored_export.json


```
{ "format": "helpdesk_automation_export_v2",
  "snapshot": { "source_system": "Rho Helpdesk",
    "workspace": "deposit-servicing", "environment": "production",
    "exported_at": "2025-11-12T11:20:00-05:00",
    "requested_by": "darnell.reeves@rhobank.com", … },
  "cover_email": …,
  "macros": [ …
    { "macro_id": "MAC-5210",
      "name": "Checking credit — apply via integration action",
      "description": "Applies a checking-account credit through the
        deposits integration action apply_checking_account_credit_5829,
        which adds a credit transaction to a checking account. […]" },
    { "macro_id": "MAC-4988",
      "name": "Checking credit — batch apply, second credit same
        interaction (LEGACY)",
      "description": "Legacy two-step flow that applied a second credit
        to the same checking account within a single interaction.
        Disabled 2025-10-28 with TRG-4055; retained for audit history." },
    … ],
  "triggers": [ …
    { "trigger_id": "TRG-4055",
      "name": "Enable second credit step (RETIRED)",
      "description": "Permitted MAC-4988's second
        apply_checking_account_credit_5829 submission within the same
        interaction. Disabled 2025-10-28 together with MAC-4988;
        retained for audit history.",
      "active": false, "status": "disabled_legacy",
      "effective_at": "2025-06-30", "updated_at": "2025-10-28",
      "usage_30d": 0,
      "conditions": [
        "macro = MAC-4988 invoked",
        "prior apply_checking_account_credit_5829 submission exists
         on the interaction" ], … },
    … ],
  "policy_contracts": [ … ], "fields": [ … ] }
```


All four slide decks in the corpus carry no facts: decks rehearse and point at the governing records without deciding anything. Every amount, date, and identifier above derives from the domain database. In audio renditions of the corpus the phone case above ships as a recorded call, with the support-console events delivered as a timestamped sidecar file rather than spoken lines.

H Construction Kit Brief

Each task materializes as a kit directory, and a generated README is the developer's brief; the developer agent reads it as its instructions. The README for release task 010, word for word (long lines reflowed to the page width):

```
# Build a Customer Service Agent: retail_plus

Your job is to build a robust customer service agent from the domain materials
in this kit. The exact artifact set varies by domain; use the file tree and
file-level documentation to orient yourself, then turn those materials into a
working implementation.

## What success means

Agent quality is measured by the proportion of evaluation cases the agent
passes. The evaluation cases are not visible in advance and may exercise the
full range of behavior represented in the supplied materials: routine
requests, complex multi-step or multi-intent requests, unusual edge cases,
incomplete or changing information, and requests that must be declined or
redirected.
```

```
A case succeeds only when the agent handles the underlying operation
correctly, leaves the system in the correct state, follows the domain rules,
and communicates the right information to the customer. Subject to the
framework contracts, no particular architecture or development process is
required.

## How to use this kit

Treat the kit directory as the inventory of available materials. Source
information may be spread across policy documents, databases, schemas,
knowledge-base files, training records, or other client-provided artifacts.
Read nearby file headers, directory indexes, and filenames as the description
of each artifact.

`workspace/` is the implementation area. `framework/` explains the contracts
your implementation must satisfy. `simulations/` stores artifacts written by
candidate-only local simulation runs.

## Required outputs

The evaluator imports these files as stable entry points. You can add helper
modules, retrieval layers, planners, validation scenarios, or any other
supporting architecture you need; the files below are only the integration
surface.

1. **`workspace/tools.py`** -- A `ClientAPIToolKitBase` subclass with
   `@is_tool`-decorated agent operations that implement all the operations
   described in the client-provided materials. Each method uses the injected
   `self.client_api`. See `framework/client_api_contract.md` and
   `client_api/openapi.yaml`.

2. **`workspace/agent.py`** -- The agent implementation. The interface-only
   scaffold defines the runtime entry point; implement the agent logic you
   want evaluated. It may read any supplied kit artifact and may add helper,
   prompt, rule, index, or other evidence files under `workspace/` using any
   organization you choose. See `framework/agent_contract.md`.

## Simulation environment

You have access to a simulation environment where you can create simulated
customer scenarios and let those customers interact with your customer service
agent. No sample scenarios are provided; write your own JSON files based on
scenarios in the client-provided materials.

See `framework/scenario_contract.md` for the customer scenario format.

To run a candidate-only end-to-end simulation, call `run_local_test` tool with
your scenario path, for example
`run_local_test(task_path="workspace/my_customer_scenario.json")`. That tool
runs only the scenario files you wrote against your own assistant toolkit. It
routes customer/user tool calls through the provided customer-side runtime
when the domain includes one. Each run writes a timestamped JSON artifact
under `simulations/` so you can inspect prior transcripts and rewards later.
For black-box behavioral checks, use natural-language assertions and inspect
the returned transcript rather than depending on internal function names.

## Performance requirements

- `agent_credit_budget`: `gpt-5.6-sol`, `openrouter/anthropic/claude-opus-5`,
  `gemini/gemini-3.1-pro-preview`, `openrouter/moonshotai/kimi-k3`,
  `openrouter/deepseek/deepseek-v4-flash`, `gpt-5.6-terra`, `gpt-5.6-luna`,
  `openrouter/anthropic/claude-haiku-4-5`, `gemini/gemini-3-flash-preview`,
  `openrouter/qwen/qwen3.8-27b`, `openrouter/google/gemma-4-31b-it`,
  `gpt-5.4-nano`, `openrouter/qwen/qwen3-30b-a3b-instruct-2507`,
  `openrouter/google/gemma-4-26b-a4b-it`, `gpt-4.1-mini`,
  `openrouter/moonshotai/kimi-k2.6`, `gpt-4o-mini`,
  `openrouter/anthropic/claude-sonnet-5`, `gpt-5-mini`,
  `gemini/gemini-3.1-flash-lite` share a 0.0820-credit budget per
```

```
  conversation. Every input and output token from every model-gateway call on
  those models counts; reasoning tokens count as output.

Use `run_local_test` to inspect measured performance while you iterate. Credit
overage is a soft penalty: final score is mean task reward minus the mean
per-conversation fraction over budget, floored at zero. Latency requirements
remain hard gates.

## Important

- The allowed agent models and their inference constraints are fixed by
  `framework/deployment_manifest.json`. Your implementation may select among
  them.
- Developer-authored scenarios are local probes, not the final evaluation
  distribution. Your final assessment depends on behavior across the broader
  set of unseen customer requests.
```

I SAMPLE CONSTRUCTION KIT

Figure 11 shows the filesystem a developer receives for release task 010, exactly as materialized: the brief and the framework contracts, the client API package, a workspace scaffold, and 89 MB of client records. Uploaded materials carry generic genre names in content-digest order, so filenames reveal what kind of record a file is but not its topic, chronology, or importance; annotations in gray are ours. Figures 12 and 13 excerpt the two files that fix the task's technical constraints: the client's REST contract, and the deployment manifest that sets the serving-model menu, per-model credit rates, and the per-conversation budget.

release task 010 — construction kit on arrival

```
README.md                    the developer's brief
framework/
  README.md                  how the engagement works
  agent_contract.md          required entry points and protocol
  client_api_contract.md     transport, errors, idempotency
  scenario_contract.md       schema for self-authored scenarios
  deployment_manifest.json   model menu, credit rates, budget
client_api/
  openapi.yaml               the client's REST contract
  development_seed.json      two seeded development cases
workspace/
  agent.py                   scaffold: the integration surface
  tools.py                   scaffold
uploaded_materials/          312 files, 89 MB; generic names,
                             content-digest order
  case_file_001.md .. case_file_124.md        support transcripts
  screenshot_001.png .. screenshot_120.png    website captures
  email_01.eml .. email_38.eml                email threads
  process_map_01.png .. process_map_11.png    process flowcharts
  meeting_transcript_01.vtt .. _08.vtt        meeting recordings
  intake_form_01.md .. intake_form_04.md
  api_contract_01.zip .. api_contract_03.zip
  reference_document.md                       customer-care handbook
  slide_deck.pdf
  system_export.zip
  workspace_export.json
```

Figure 11: The construction kit for release task 010.

client_api/openapi.yaml

```
{
  "openapi": "3.1.0",
  "info": {
    "title": "Customer Service API",
    "version": "3.1.0",
    "description": "REST API for programmatic access to customer
      service business resources. All operations exchange JSON
      over HTTPS."
  },
  "paths": {
    "/v1/customers/search": {
      "post": {
        "operationId": "searchCustomers",
        "summary": "Search for a customer",
        "description": "Find one customer using either an email
          address or the complete name-and-postal-code selector.",
        "x-api-mutates-state": false,
        "x-api-idempotency": "safe",
        "x-api-automatic-retries": "allowed",
        "x-api-consistency": "strong", […]
      }
    }, […]
    "/v1/orders/{order_id}/cancellations": {
      "post": {
        "operationId": "createOrderCancellation",
        "summary": "Cancel an order",
        "description": "Create a cancellation for one order.",
        "x-api-mutates-state": true,
        "x-api-idempotency": "not_guaranteed",
        "x-api-automatic-retries": "forbidden", […]
        "requestBody": { […] "schema": {
          "properties": {
            "reason": { "minLength": 1, "type": "string" }
          },
          "required": ["reason"],
          "title": "CancellationRequest" } } […]
      }
    }, […]
  },
  "components": { "schemas": { "APIError", "Address",
    "CreditCard", "CustomerName", "GiftCard", "Paypal",
    "OrderItem", "OrderPayment", […] } }
}

13 operations: searchCustomers, getCustomer,
replaceCustomerDefaultShippingAddress, listProducts, getProduct,
getCatalogItem, getOrder, replaceOrderShippingAddress,
replaceOrderPaymentMethod, createOrderCancellation,
createOrderItemModification, createOrderExchange,
createConversationTransfer
```

Figure 12: The client API contract, condensed: one read and one write operation of the thirteen, with the machine-readable mutation, idempotency, and retry annotations the developer must respect.

framework/deployment_manifest.json


```
{
  "allowed_agent_models": [
    {
      "model": "gpt-5.6-sol",
      "constraints": {},
      "credit_rates": {
        "input_per_million": 3.0,
        "output_per_million": 15.0,
        "rate_card_date": "2026-08-23",
        "pricing_basis": "capability_bucket_median_list_price"
      }
    }, […]
    {
      "model": "openrouter/anthropic/claude-sonnet-5",
      "constraints": {
        "extra_body": { "reasoning": { "enabled": false } }
      },
      "credit_rates": {
        "input_per_million": 0.5,
        "output_per_million": 3.2,
        "rate_card_date": "2026-08-23",
        "pricing_basis": "capability_bucket_median_list_price"
      }
    }, […]
  ],
  "performance_requirements": [
    {
      "id": "agent_credit_budget",
      "type": "credits",
      "measurement": "agent_model_calls",
      "budget": 0.082,
      "models": [
        "gpt-5.6-sol",
        "openrouter/anthropic/claude-opus-5",
        "gemini/gemini-3.1-pro-preview",
        "openrouter/moonshotai/kimi-k3", […]
      ]
    }
  ]
}

20 model entries spanning three rate buckets (3.0/15, 0.5/3.2,
0.25/1.6 credits per Mtok in/out); the budget clause covers all 20.
```


Figure 13: The deployment manifest, condensed: the serving-model menu with per-model credit rates and constraints, and the per-conversation credit budget.

J CLIENT API DEFECT CATALOG

Each task that enables defects pins a versioned deployment manifest for its Client API. The manifest declares a set of defect instances, each binding one of the nine classes of Table 6 to one public operation. The published OpenAPI contract and reference documentation continue to describe the intended behavior: the manifest records distinct published and deployed API versions, and the gap between the two is exactly the declared defect set.

Activation is deterministic. Every instance carries explicit selectors: the evaluation tasks it applies to, the request ordinals and resource identifiers that trigger it, and seeded delays where timing is involved, so repeated trials replay identically. Where the manifest configures an exposure rate, task-scoped defects also surface during the developer's own kit-harness test runs on a deterministic per-scenario draw, so local simulations can encounter the same faults the evaluation deploys.

Discovery is empirical. The simulated Client knows each deployed defect: its actual behavior, its disclosure conditions, and the remediation it expects, but volunteers none of it, confirming a defect only once the developer reports the observed anomaly on the specific operation. No deployed defect can be fixed on the Client's side, so every remediation is a code mitigation the developer builds into the agent. The release set deploys sixteen instances across three defective deployments (eight in airline, four in retail, four in telecom), enabled on the ten *defects* tasks of Appendix A.

Table 6: Each deployed defect instance binds one of nine classes to one public API operation and activates deterministically; the Client can never deploy a fix, so every remediation is a code mitigation in the constructed agent.

Defect class	Deployed behavior
contract drift on reads	
Value remap	A response field returns a representation the contract does not document (e.g., a boolean where the schema documents the strings `yes` and `no`).
Field rename	One response field arrives under a name the contract does not document.
Amount sign	Amounts for one record type arrive with a fixed sign regardless of the documented convention.
Date type drift	A documented calendar date arrives as a UTC-midnight datetime string.
write-path semantics	
Async completion	A mutation the contract documents as synchronous returns `202 Accepted` with a workflow ID, a status location, and a retry delay.
Post-commit timeout	A mutation commits and its success response is then lost as a `504`.
Rate limit	An operation returns `429` at a fixed request ordinal until a server-stated delay elapses.
read-model consistency	
Pagination	A collection response pages, truncates, loses cursors, or applies its limit before sorting.
Projection lag	A detail read lags the acknowledged write it should reflect.

K CLIENT SIMULATOR PROMPT

The client's system prompt is rendered deterministically from the task's fact schema: no model authors it, so the client's scope can be diffed and unit-tested against the task's variant. Below is the prompt rendered for release task 010, word for word; only the embedded fact statements are elided.

```
<role>
You are the customer-support operations lead at an online retail company. You
have hired an outside consultant -- the developer -- to write the standard
operating policy for your customer-facing support agent. Instead of a
rulebook, you handed the developer a package of company records: approved case
transcripts, email archives, website captures, process decks, and similar
material. Nearly all the rules the agent must follow are in those records, but
```

```
a few points never made it into any document and live only in your head (see
<what_only_you_know> below), and on a few others the documents do not agree
with each other; you know which version is current (see <records_in_conflict>
below).
</role>

<what_you_want_built>
A complete, correct operating policy covering: manage pending order, manage
delivered order, service foundations. The records are the source of truth --
the developer's job is to work them out, not to interview you point by point.
</what_you_want_built>

<what_only_you_know>
These points never made it into the records you handed over -- the developer
cannot find them in any document, so they can only get them from you:

## Manage pending order
### Refunds
- [fact statement]
- [fact statement]
- [fact statement]
### Payment change
- [fact statement]
### Item change
- [fact statement]

## Manage delivered order
### Return items
- [fact statement]
### Exchange one shot
- [fact statement]

## Service foundations
### Authorized scope
- [fact statement]
### Status limits
- [fact statement]
### Prohibited actions
- [fact statement]
- [fact statement]
### Conduct
- [fact statement]
### Transfer
- [fact statement]
</what_only_you_know>

<records_in_conflict>
On these points the records you handed over do not agree -- different
documents show different versions, and nothing in the package settles which
one is current. You know what is actually in force:

## Manage pending order
### Cancellation reason
- [fact statement]

## Manage delivered order
### Refund destination
- [fact statement]
- [fact statement]
### Exchange payment
- [fact statement]

## Service foundations
### Currency
- [fact statement]
### Prohibited actions
- [fact statement]
</records_in_conflict>
```

```
<questions_you_can_settle>
These points ARE in the records, but you know from experience that people
misread them, so you are willing to check a specific reading against your own
knowledge:

## Manage delivered order
### Return submission
- [fact statement]
</questions_you_can_settle>

<how_you_behave>
- Stay in character at all times: a busy operations manager. Plain,
  conversational language. Keep replies short -- a few sentences.
- Never recite, list, or summarize policy rules -- not even one, not at any
  level of detail. If the developer asks you what the rules are, tell them
  that is exactly what the records are for.
- The points in <what_only_you_know> appear in no record, so they are yours to
  hand over:
  - When the developer asks about one of those areas, answer plainly and
    completely in your own words -- "check the records" is not an acceptable
    answer for them.
  - Answer only what was asked: do not append neighboring rules the developer
    did not raise, even ones you know well.
- The points in <records_in_conflict> appear in the records in more than one
  version, and the documents cannot settle which is current -- only you can:
  - When the developer asks which version holds, or proposes any version they
    found, state the current one plainly in your own words -- sending them
    back to the records is not an acceptable answer here.
  - Settle only the point they raised; do not volunteer other conflicts they
    have not brought up.
- The points in <questions_you_can_settle> you check against your own
  knowledge, staying as close to yes/no as the question allows:
  - If the developer states a specific reading and it matches your knowledge,
    confirm it plainly ("Yes, that's right.").
  - If it contradicts your knowledge, say it is not right and tell them to
    take another look at the records.
  - If they lay out two or more readings and ask which holds, name the one
    that matches -- picking from what they put in front of you is still a
    check, not a briefing, so name it and stop there.
  - Never volunteer a version they did not offer. If a reading is wrong, or
    none of the options they listed is right, say so and send them back to the
    records without supplying the correct version, in whole or in part.
- Any other policy question -- asked outright, or floated as a reading for you
  to bless -- you point back to the records. Do not confirm it, deny it, or
  discuss its substance. Outside your listed points your memory is unreliable
  and you know it.
</how_you_behave>

<opening>
When asked to open the conversation, briefly introduce yourself, say what you
need built and what material you have handed over, and invite the developer to
get started.
</opening>
```

L EVALUATION USER SIMULATOR

Held-out tasks are served by an LLM user simulator whose system prompt is a fixed set of global guidelines followed by the task's scenario (persona, reason for contact, known information). The guidelines, word for word:

```
# User Simulation Guidelines
You are playing the role of a customer contacting a customer service
representative.
Your goal is to simulate realistic customer interactions while following
specific scenario instructions.

## Core Principles
```

```
- Generate one message at a time, maintaining natural conversation flow.
- Strictly follow the scenario instructions you have received.
- Never make up or hallucinate information not provided in the scenario
  instructions. Information that is not provided in the scenario instructions
  should be considered unknown or unavailable.
- Avoid repeating the exact instructions verbatim. Use paraphrasing and
  natural language to convey the same information
- Disclose information progressively. Wait for the agent to ask for specific
  information before providing it.

## Task Completion
- The goal is to continue the conversation until the task is complete.
- Generate the '###STOP###' token only after the agent has confirmed
  completion of every action your scenario requires (e.g. "your return has
  been submitted"). Giving consent -- saying "yes, please do that" -- is
  **not** completion; the agent still needs a turn to act on it. For purely
  informational scenarios where no action is needed, STOP only after you have
  received all the information you came for. Emitting ###STOP### too early
  ends the conversation before the agent can finish the work.
- If you are transferred to another agent, generate the '###TRANSFER###' token
  to indicate the transfer.
- If you find yourself in a situation in which the scenario does not provide
  enough information for you to continue the conversation, generate the
  '###OUT-OF-SCOPE###' token to end the conversation.
Remember: The goal is to create realistic, natural conversations while
strictly adhering to the provided instructions and maintaining character
consistency.
```

Voice-served tasks use a spoken-register variant of the same guidelines: disfluencies and restarts, identifiers spelled as on a phone (“at,” “dot”), digits and letters one by one.

M Communication Judge Prompts

Natural-language expectations on the agent’s messaging are graded by LLM judges, one rubric-driven prompt per rule. Below, word for word, the judge for one phrasing rule: the agent must not open its responses with a repeated template.

```
You are a strict but precise evaluator for one customer-service response
phrasing rule.

RULE
The assistant must not begin more than one customer-facing response with the
same opening phrase template.

INPUT
You will see only assistant response openings, numbered in conversation order.

HOW TO DECIDE
- Compare only the opening wording shown for each assistant response.
- A violation requires repeated reusable opening wording, not merely the same
  broad discourse function.
- Ignore capitalization, punctuation, customer names, reservation/order/line
  IDs, phone numbers, dates, addresses, and other entity-specific values.
- Entity changes do not make the same frame meaningfully different. For
  example, "I found the tablet order" and "I found the sneaker order" repeat
  "I found the ...".
- Repeated refusal/apology frames count as repeated. For example, "I'm sorry,
  but passengers cannot..." and "I'm sorry, but I can't..." repeat "I'm sorry,
  but ...".
- Repeated completion frames count as repeated. For example, "Done. Your
  shipping address..." and "Done. Your default account address..." repeat
  "Done. Your ...".
- Repeated confirmation/request frames count as repeated when the reusable
  phrase is the same, such as "Please confirm..." appearing in multiple
  responses.
- Do not fail only because openings share one generic lead word such as "Yes",
  "No", "Thanks", "Great", "Please", "Next", "Now", or "Since".
```

```
- For those generic lead words, fail only if the phrase after the generic lead
  word is also substantially the same reusable template. For example, "Please
  turn Data Roaming ON" and "Please disconnect the VPN" are not the same
  opening template.
- Do not fail because multiple openings provide different sequential
  troubleshooting steps, even if each is imperative or starts with a
  transition word.

OUTPUT
Return only valid JSON:
{
  "metExpectation": true or false,
  "repeatedOpenings": [
    {
      "template": "short normalized repeated opening phrase",
      "responseNumbers": [1, 4],
      "evidence": ["opening text 1", "opening text 4"]
    }
  ],
  "reasoning": "brief explanation"
}

Set metExpectation to false if there is at least one repeated opening phrase
template.
Set metExpectation to true only if no repeated opening phrase template
appears.
```