RESEARCH PAPER

Overseer of the Overseer: Shipping Sixteen Projects in Four Days with a Routed Crew of Language-Model Agents

A case study in orchestrator/worker/subagent routing under a metered budget.

JULY 2026

ABSTRACT

This paper documents a four-day production run, July 8 to 11, 2026, in which one person shipped sixteen real projects while mostly away from a keyboard. The work spanned client websites, infrastructure operations, a family member's personal assistant, content pipelines, and personal admin. The operating model was not a single powerful agent but a crew: one long-lived orchestrator that plans, dispatches, and verifies but never builds; workers that each own one project; and short-lived subagents routed to the cheapest model that clears the task. Two metered Claude accounts ran in parallel, coordinated through an append-only status log and reached from the road by scannable phone notifications and a tap-to-approve gate for anything irreversible. The human sat above the orchestrator, setting direction and tapping Approve or Deny. This report describes the architecture component by component, presents the campaign as data with real wall-clock samples (including a peak of eighteen concurrent agent sessions), and treats the run's failures as first-class findings. Five of those failures, from a confabulated rogue-orchestrator report to cross-account deliverables that rendered for nobody, drove concrete design changes. No token metering existed per tier during the run, so all cost claims stay qualitative. This is a case study of one operating model, not a benchmark.

1 Introduction

A language-model agent working alone runs into three walls. The first is context. A single session accumulates the transcript of everything it has read and done, and past some length it either compacts (losing detail) or stalls. A day of real work across a dozen projects does not fit in one window. The second is cost shape. The most capable models are also the most expensive per token, and a large share of any real project is mechanical: reading files, running greps, restarting a service, checking a status. Paying top-tier rates to do a two-command chore is waste, and under a metered budget that waste is measured directly in how much less can ship. The third is trust. An agent's report about its own work is not evidence. It will say a page renders when it does not, or claim it has a tool it does not have, with the same fluent confidence it uses when it is right.

These three walls point at the same opening. If no single session can hold the whole day, split the day across many sessions. If mechanical work should not run on expensive models, route each piece of work to a model sized for it. If self-reports cannot be trusted, make verification a separate step done by a party that did not do the building. That is orchestration, and it is not a new idea. What has been missing from the public record is an honest, reproducible account of running it in production, on real deadlines, by one person, with the failures left in.

This paper is that account. Over four days one operator shipped sixteen projects using a specific operating model: a single orchestrator agent that never builds anything itself, delegating to right-sized worker models across two metered accounts in parallel, with retrieval-first recall to keep context lean, a phone-based approval gate for anything irreversible, watchdog scripts for usage caps, strict routing of heavy compute onto the right machines, and independent verification of every claimed "done." The contribution is threefold. It is a documented operating model that another practitioner could rebuild. It is a set of real numbers from a genuine run, wall-clock timings and concurrency peaks rather than a demo. And it is a catalog of the ways the system broke, each paired with the design change it forced. The last part matters most. A method paper that only shows the parts that worked teaches less than one that shows where the confident-sounding agent lied and what had to change.

2 Related work

The system draws on several established lines of work without inventing new theory in any of them.

Retrieval-augmented generation, introduced by Lewis et al. (2020), pairs a generator with a retriever so the model pulls in external documents rather than relying on parametric memory. The recall layer here is a direct application: query an index over project docs, infrastructure notes, and past decisions before reading any file.

Reasoning-and-acting loops, formalized as ReAct by Yao et al. (2023), interleave a model's reasoning with tool calls so it can observe results and adjust. Every worker and subagent here runs such a loop, calling tools and reacting to what comes back.

Tool use itself has a training-side treatment in Toolformer (Schick et al., 2023), which taught models when to call external APIs. In this run tools are supplied at inference time through an interface layer rather than learned, but the same premise holds: a model is far more useful with hands than without.

Multi-agent orchestration is the closest neighbor. AutoGen (Wu et al., 2023) provides a framework for conversations among multiple agents; HuggingGPT (Shen et al., 2023) uses a controller model to plan and dispatch subtasks to specialized models. The overseer/delegate split described here is in that family, with a stricter rule (the orchestrator never executes) and a routing decision made per subtask.

Model routing and cascades address the cost shape directly. FrugalGPT (Chen et al., 2023) showed that routing queries across a cascade of models by difficulty can cut cost sharply while holding quality. The right-model routing rubric here is a hand-authored version of the same idea, chosen because the budget is metered in fixed windows.

Software-engineering agents that act on real repositories, such as SWE-agent (Yang et al., 2024), demonstrate agents editing code and running tests against real issues. The workers in this run do that kind of work, but each is wrapped in an independent verification pass rather than trusted on its own report.

Agent memory and coordination echo Generative Agents (Park et al., 2023), which gave agents an external memory stream to persist and retrieve experience across time. The record layer and the append-only status log play that role: externalized state that survives a context reset.

Human oversight of automated systems has a longer lineage. Christiano et al. (2017) put a human in the loop to steer reinforcement learning from preferences, and Amodei et al. (2016) laid out concrete safety problems including the value of human checkpoints on irreversible actions. The approval gate is a plain engineering instance of that principle: block on a human tap before anything that cannot be undone.

The system combines these ideas rather than extending any of them. What is new is not a technique but a documented, reproducible way to run them together under real constraints, with the failure analysis attached.

3 Method

The architecture has nine components. Figure 1 shows how they fit: the human at the top, the orchestrator below, workers fanned out under it, subagents under each worker, and the shared services (retrieval, status log, approval gate, watchers) threaded through.

3.1 The overseer/delegate split

The orchestrator thinks, plans, dispatches, and verifies. It does not edit code, render video, or take screenshots. This is a hard rule, not a preference. The orchestrator's value is that it holds design direction and judgment across the whole run, and every token it spends reading a file it could have delegated is a token not spent on judgment. So it stays clean and pushes hands-on work down.

The rule is fractal. Each worker owns one project and is itself an overseer for that project, delegating hands-on work to subagents in small batches, three to four concurrent at most (Figure 2). A worker plans its project, spawns subagents to do the reading and editing and testing, reads their results, and decides what happens next. It does not do the mechanical work itself any more than the orchestrator does. The batch cap of three to four exists because a worker that fans out too wide loses the same thing the orchestrator would: the ability to hold the whole picture while the pieces move.

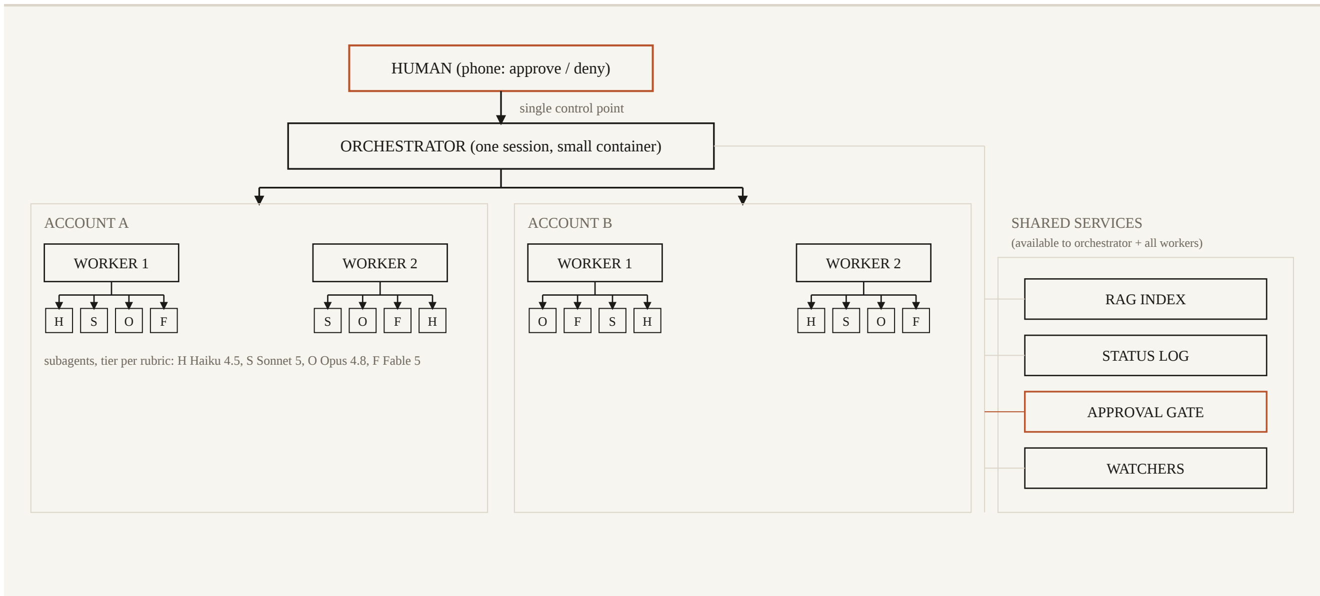

Figure 1. One orchestrator directs two worker accounts, each fanning out to tier-routed subagents, with shared services on the side and a single human approval point at the top.

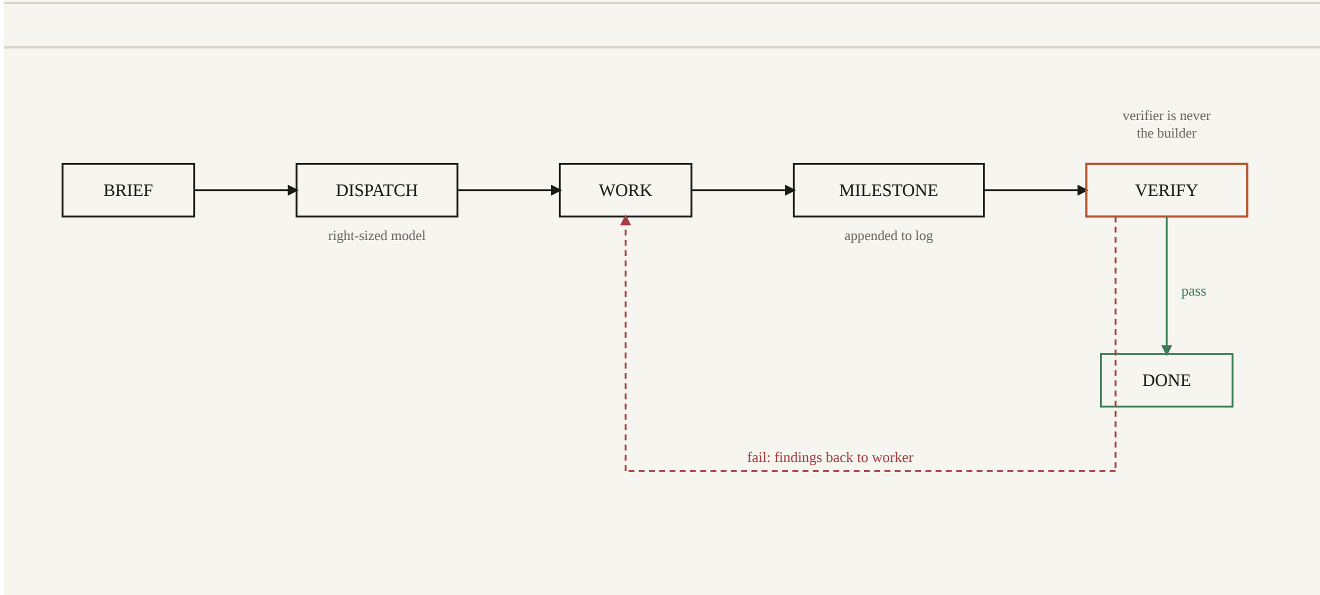

Figure 2. Each unit of work moves through dispatch, execution, and a logged milestone, then an independent verifier passes it to done or sends it back with findings.

3.2 Right-model routing

Every subtask is assigned a model sized to it (Figure 3). The tiers used, by public model name:

- Haiku 4.5 for trivial mechanical work: file reads, greps, service restarts, status checks.
- Sonnet 5 for bounded technical chores: diagnosis, config edits, focused fixes, summarizing.
- Opus 4.8 for harder reasoning, writing, and judgment.
- Fable 5 for big autonomous builds, design work, and multi-hour missions.

The rubric runs both directions. Do not spend a Fable agent on a two-command chore. Do not send Haiku at an architectural decision. The orchestrator itself stays on the top tier because it carries the design direction, but it extracts its taste through cheap hands given very specific instructions. A precise instruction to a small model often beats a vague one to a large model, and it costs far less. The motivation is not abstract frugality. Both accounts are metered in five-hour windows, so top-tier tokens spent on mechanical work reduce, directly and immediately, how much can ship before the window resets.

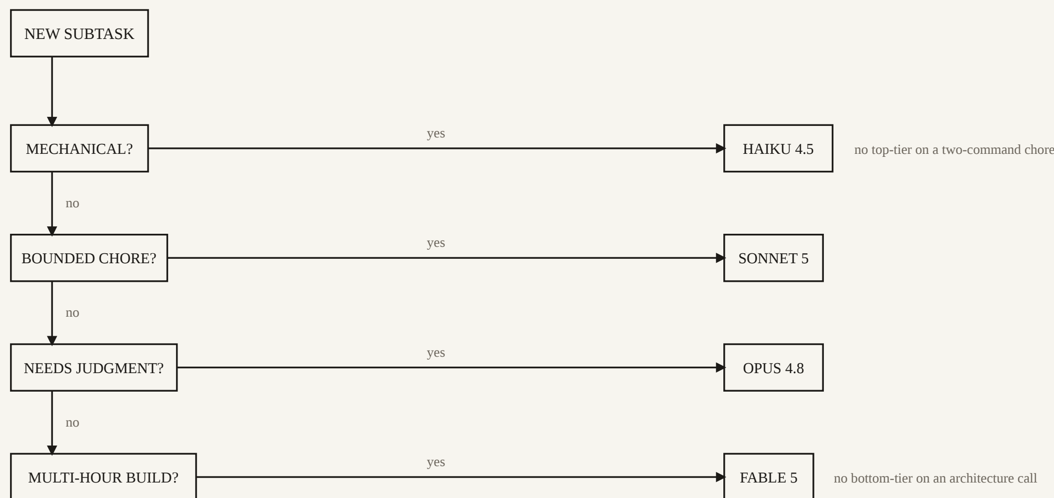

Figure 3. A new subtask falls through mechanical, bounded, judgment, and build questions until it lands on the cheapest model that clears the bar.

3.3 Three-layer retrieval: recall, navigate, record

Context stays lean by querying the right layer before touching files (Figure 4).

Recall comes first. Before reading any file, query a local RAG index over docs, infrastructure notes, and project history. Open a specific file only when recall comes up short, and never sweep a file tree. Retrieval misses are logged so the index improves over time. This is the single biggest lever on context size: a good recall answer is a few hundred tokens where a directory sweep is tens of thousands.

Navigate is for live code work. Instead of grepping trees to understand a codebase, workers use a code knowledge graph and symbol tools to find definitions and trace blast radius. The question "what calls this function" gets answered from the graph, not from reading every candidate file.

Record closes the loop. Durable decisions and handoffs are written to a memory graph and to per-topic memory files. The point is recovery: the next session, or the same session after its context compacts, can pick up state cheaply instead of re-deriving it.

Figure 4. A query hits recall first and only falls through to a single file read on a miss, while navigate and record serve code work and durable state on their own tracks.

3.4 The approval gate

Anything irreversible, outward-facing, or high blast-radius passes through a command-line gate. The `propose` tool sends the human a phone push carrying Approve and Deny buttons and blocks the agent until he taps (Figure 5, left). The exit-code contract is strict: 0 means approved, 1 means denied, 2 means timeout, and timeout is treated as not approved. There is no default-yes. If the human is asleep and the window elapses, nothing ships.

This held in production. Two overnight deploy proposals timed out while the operator slept. They were re-sent with longer windows and shipped only on the morning Approve tap. The gate did exactly what it was built to do: it kept an unattended agent from pushing to production on its own schedule, and it cost only a few hours of latency to do so.

3.5 Watchers

Detached watchdog scripts monitor each worker session (Figure 5, right). On a real usage-cap hit, a watcher fires an urgent phone push and writes a resume file so the same session continues after the window resets. It never restarts the session, because a restart would discard the work in progress. On sustained idle, the watcher sends a "finished" push so the operator knows the project has landed or stalled. Watchers observe only. They never inject input into a session and never assume state they have not seen. Their job is to turn silent conditions into a notification, not to act.

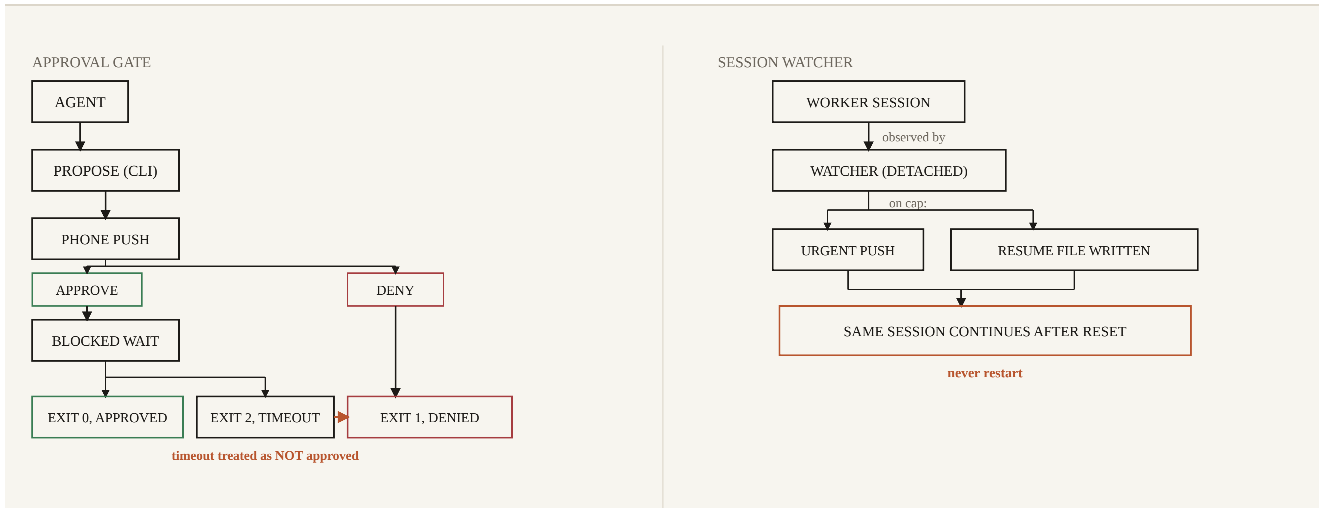

Figure 5. The gate blocks on a phone tap and treats a timeout as a denial, while a detached watcher resumes the same session after a usage cap instead of restarting it.

3.6 Compute routing

Machines are matched to workloads. The orchestration container never builds. This rule was written in blood: a video render once spawned forty-six headless-browser processes on that small container and nearly took it down. So heavy builds, tests, and headless-browser verification go to a heavier container, the forge. GPU work (video, image generation) goes to a Windows desktop with an RTX-class GPU. During this run that GPU box was under a hard rule of its own: never reboot it remotely, because a failed boot could not be recovered from the road. The forge runs one heavy job at a time on its single disk; two concurrent heavy jobs once spiked node load to 330, which is why serialization on that box is enforced rather than suggested.

3.7 Two-account parallelism

Two metered accounts run in parallel: the operator's own Claude account and his dad's, each with a rolling five-hour usage window. The rule is one project per account at a time, parallel across the two. Work shifts to whichever account has headroom, and remaining budget is sometimes spent deliberately just before a window resets rather than left on the table. This doubles the throughput ceiling without touching a single model's rate limits. The measured peak was eighteen live agent sessions at once, eight on one account and ten on the other.

3.8 Verify-before-done

"Should work" is not done. Every claimed completion is exercised by an independent subagent, one that did not do the building. That subagent loads the page as the real user would, screenshots it on the forge, drives the UI with a browser automation tool, and hits the live endpoint. This is not a formality. On the budget-system upgrade, an independent verify pass found three real bugs in cleanup scripts plus a gap where a panel failed to render, all of which the builder had reported as done. Those went back as fixes before the project was allowed to close. The extra pass costs tokens and time, roughly a second run per deliverable, and it earns that cost the first time it catches a builder's confident mistake.

3.9 The status protocol

A system whose agents share no memory still has to coordinate. It does so through one append-only log. Every worker writes one-line milestones in a fixed format: a UTC timestamp, the session name, a tag (MILESTONE, BLOCKER, NEEDS-HUMAN, or DONE), and one line of text. Workers also push scannable phone notifications at milestones so the operator can track progress without reading logs. The orchestrator tails the shared log to catch up after its own context window compacts. This is the backbone of the whole run: when the orchestrator loses its short-term memory to compaction, the log is how it recovers what everyone is doing.

4 The campaign

The run covered about four days, July 8 to 11, 2026. At the moment the mid-run record was cut, late on the night of July 10, thirteen projects had shipped. Three more finished overnight into July 11: a UI navigation and scoped-feedback feature, a set of platform-compliance pages plus an app icon generated on the GPU box, and a researched project plan. Two more were in flight when the record was taken. That is sixteen shipped across the window. One tool built during the run, the approval gate itself, is counted as part of the method rather than the campaign output, which is why the category chart in Figure 7 shows thirteen at the record cut.

The work sorts into four categories, described by category rather than by client.

Client and revenue work. A business admin console with lead scoring, outreach templates, and booking. An online course offering built from a client's own transcribed voice memos. An evaluation of an AI framework with an architecture recommendation and a client-safe summary package. A content-pipeline rollout across three social accounts.

Infrastructure and operations. A full infra health audit, which found and rotated an exposed TLS key. A live usage dashboard for both accounts. A wall-display news kiosk. A framebuffer status display on a spare monitor. A site maintainer tracker. The approval gate itself, plus a clean per-project notification-channel scheme.

A family member's assistant. An agent ecosystem on its own container, where ten voice memos were transcribed and nine requests were folded into six new skills, plus a budgeting-system upgrade with a weekly check-in cadence, debt-snowball logic, and streaks.

Personal admin. 20,367 promotional emails archived, real bills reconciled, and seventeen stale calendar events cleared.

The wall-clock samples below come straight from the shared status log (UTC, July 10 to 11). They are single sessions, not aggregates.

PROJECT	START	DONE	DURATION	NOTES
Online course offering	22:02	22:34	32 min	3 voice memos transcribed on the forge CPU; mobile and desktop screenshot-verified

PROJECT	START	DONE	DURATION	NOTES
Budget system upgrade	22:03	22:33	30 min	Ran in parallel on the second account; builder plus independent verifier; verify found 3 bugs, fixed before done
AI-framework evaluation	kickoff	report	~22 min	Written and verified report
Usage dashboard	00:22	01:04	42 min	Mid-run scope addition; recovered from a real API rate-limit outage
Researched project plan	04:42	05:05	23 min	Three research subagents in parallel
UI feature with approval gates	(built 04:10)	07:19	overnight	Deploy proposal timed out twice overnight; approved and shipped 07:19; production-verified 07:28

Two of these overlap in time on purpose. The online course offering (22:02 to 22:34) and the budget upgrade (22:03 to 22:33) ran on different accounts in the same half hour. That overlap is the two-account parallelism paying off in the log rather than in theory.

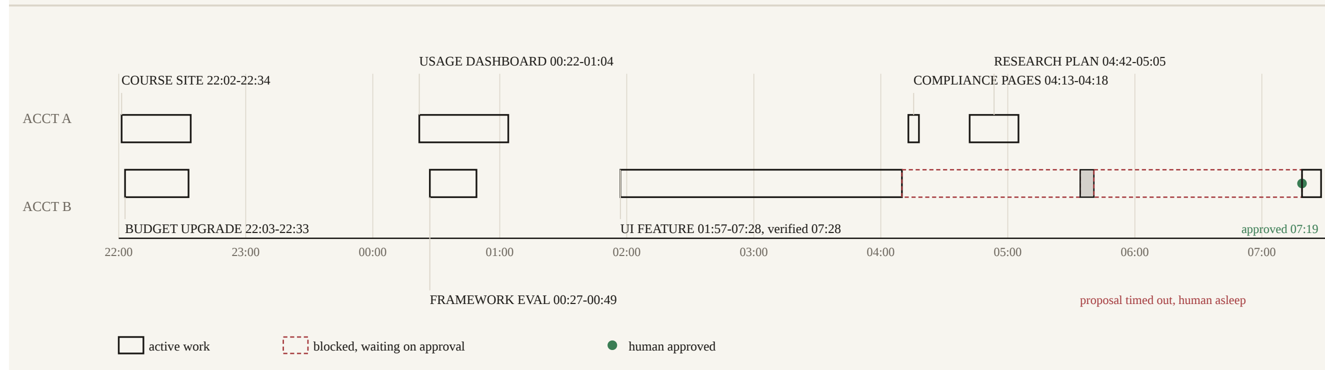

Figure 6. Real timestamps from the shared status log show two accounts running work in parallel through the night, with the one overnight approval gate visible as a stalled bar that resolves at the morning tap.

5 Results and analysis

5.1 Throughput and parallelism

Sixteen shipped projects in four days, run by one person mostly away from a keyboard, is the headline figure. The mechanism behind it shows up in two places in the data. The wall-clock samples show medium-size projects landing in twenty to forty-five minutes each, verification included. The concurrency peak shows how many of those ran at once: eighteen live agent sessions, eight on one account and ten on the other (Figure 7 summarizes the run; Figure 6 shows the overnight swimlane where sessions overlap). No single session did all sixteen projects. Many small, verified sessions did, layered across two accounts and coordinated through the log.

The overnight swimlane is the clearest evidence that this was real parallelism and not sequential work relabeled. Between roughly 22:00 on July 10 and 07:30 on July 11, the log shows overlapping build sessions, research fan-outs, and gated deploys sitting idle waiting for a morning tap, all interleaved. The 22:02 and 22:03 starts on separate accounts are the simplest example; the 04:42 researched plan with three parallel research subagents is another.

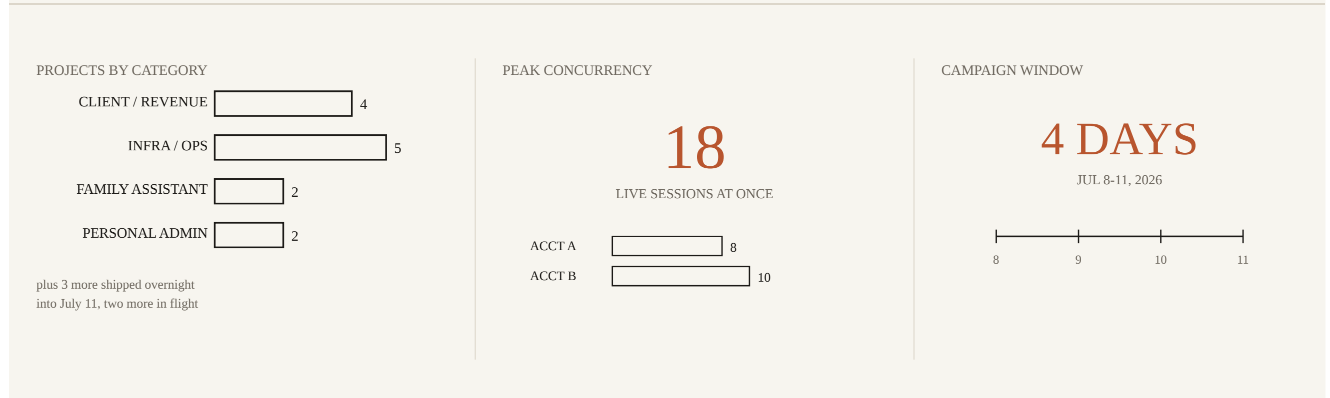

Figure 7. Thirteen projects at the mid-run record, with three more finished overnight, spanned client, infrastructure, family, and personal work; concurrency peaked at eighteen live sessions across the four-day window.

5.2 Routing economics

Right-model routing is a core design choice, and its motivation (spend cheap tokens on cheap work so top-tier budget goes to judgment) is sound. But we did not meter tokens per tier during the run. There is no per-model token count, so there is no quantitative cost claim to make. Describing the routing as saving a specific percentage would be inventing a number. What can be said is qualitative and still worth saying: the rubric was applied consistently, mechanical work was pushed to Haiku and Sonnet, and the top tier was reserved for builds, design, and the orchestrator's own judgment. Whether that produced the cost savings the design predicts is a question the instrumentation of this run cannot answer. Per-tier metering is future work, and Section 6 says what it would take.

5.3 Failure modes as findings

The run's failures are reported here as findings, each with what happened, the root cause, the design change adopted, and the general lesson.

F1 The phantom rogue orchestrator. A monitoring agent reported that a rogue orchestrator session existed and pointed at a suspicious file. Direct inspection found neither: no such session, no such file. The agent had confabulated its own reconnaissance and reported the confabulation as fact. Separately, an agent asked whether it had email tools available answered "yes, ten tools" and was bluffing; none of them worked. Root cause: a language model's self-report is generated text, not an observation, and it is produced with equal fluency whether or not it corresponds to reality. Design change: verification became a first-class pipeline stage rather than an afterthought, and capability checks were made behavioral. Do not ask a model whether it has a tool; have it call the tool and observe the result. General lesson: a subagent's self-report is a hypothesis, and the pipeline must contain a step that turns hypotheses into checked facts.

F2 The dead email connector. An account-level email integration worked in interactive sessions but was silently dead in headless ones. Combined with F1-style confident self-reports about email being available, this burned hours across sessions before the workaround, plain IMAP with an app password, was adopted. Root cause: the integration depended on interactive authentication that simply was not present in the automation runtime, and it failed by returning nothing useful rather than by erroring loudly. Design change: probe the actual call path in the actual runtime before building on any integration. General lesson: integrations that lean on interactive auth fail invisibly under automation, and "it works when I use it by hand" is not evidence that it works headless.

F3 The em-dash that broke a notification. An approval-gate push once failed to send because an em-dash in the message broke the notification path. A single character silently disabled the system's one human control point. Root cause: the control-plane transport did not handle the character, and the failure was silent. Design change: control-plane messages now use the most conservative encoding available, and control paths are tested with realistic payloads rather than the word "test." General lesson: the human control point is the most safety-critical path in the system, so it deserves the most defensive encoding and the most realistic testing, not the least.

F4 False cap reads from stale scrollback. The usage-cap watchers originally scraped terminal scrollback and sometimes read an old cap banner as current, flagging healthy sessions as capped. A confident belief that the usage windows anchor to a fixed minute also turned out to be wrong. Fix: switch to the actual usage API, which returns exact reset timestamps. That API then rate-limited under sixty-second polling, producing a real 429 outage, and needed caching and backoff to behave. Root cause: scraping rendered UI for machine state is fragile in two ways at once, stale content and layout drift, and the replacement API had its own discipline requirements. Design change: read state from the API with exact timestamps, cache it, and back off. General lesson: rendered UI is not a machine-readable state source, and even the correct API is not free to poll; polling needs a budget too.

F5 Cross-account invisible deliverables. Deliverables published as private artifacts from the second account did not render for the operator, because they were private to the publishing account, and links pushed to his phone opened to nothing. Every second-account deliverable had to be republished from the main account until the durable fix landed: a public, unlisted static site as the canonical home for deliverables, reachable from any device with no login. Root cause: the delivery surface had an account-scoped visibility boundary that the design had not accounted for. Design change: one login-free canonical delivery surface for everything. General lesson: the delivery surface is part of the system, and links must be tested as the actual reader on the actual device, not assumed to resolve because they resolve for the author.

F6 Mechanical paper cuts. Three smaller ones are worth recording. The terminal input path needed a submit-twice pattern because messages queued but did not send on the first try. A process-kill pattern once matched its own search string and killed the wrong thing, exiting 144. And worker sessions take about a minute to boot their tool servers, so their status has to be polled rather than assumed ready. None of these was fatal, but each cost time until it was understood, and together they are a reminder that the glue between agents is where the small, silent failures live.

6 Discussion

This pattern is worth its overhead under specific conditions. It pays when there are many parallelizable, medium-size projects rather than one deep one, because the throughput comes from running many short verified sessions at once. It pays when a human is reachable, because the approval gate and the notifications assume someone can tap Approve within a reasonable window. And it pays when the budget is metered, because the whole routing discipline exists to stretch a fixed window, and without metering the discipline is solving a problem you do not have.

It breaks in the mirror-image conditions. Gate latency is real: the overnight proposals sat for hours because the one human who could approve them was asleep. For work that must ship on a fixed clock without a human awake, the gate is a bottleneck, and loosening it trades safety for speed in exactly the place you least want to. Verification overhead is real: a second independent pass per deliverable costs tokens and time, and on a run where nothing ever slips through, that cost buys insurance you did not end up needing. Orchestrator compaction is real: the top-level session loses its short-term memory and has to rebuild state from the status log and state files, which costs re-reading and can drop nuance the log did not capture. And the whole design leans on one human's phone being reachable. A single point of human dependence is a single point of failure, and this run had exactly one.

There is a genuine tradeoff between oversight and cost. Every approval gate and every verification pass makes the system safer and slower and more expensive. The run chose heavy oversight and accepted the latency, which was the right call for outward-facing client work where a bad deploy is worse than a slow one. A different workload might tune it the other way.

Making the cost claims quantitative is the clearest piece of future work. It would take per-tier token accounting: log every subagent's model and token count, tag it by task class, and total by tier per window. With that, the routing hypothesis (cheap hands on mechanical work saves top-tier budget for judgment) becomes measurable, and the savings can be reported as a number instead of a design intention. The instrumentation is not hard. It simply did not exist during this run.

7 Conclusion

Sixteen projects in four days, run by one person mostly away from a keyboard, came from an operating model rather than a bigger model. One orchestrator that never builds, workers that each own a project, subagents routed to the cheapest model that clears the bar, two metered accounts in parallel, retrieval before reading, a phone tap before anything irreversible, and an independent check on every claimed "done." The failures were as instructive as the throughput: a confabulated report, a silently dead integration, one character that broke the control plane, brittle state-scraping, and deliverables that rendered for nobody, each of which forced a specific fix. The pattern is not a benchmark and makes no claim against other systems. It is one reproducible way to get real work shipped under real constraints, with the seams left visible so the next person can build it better.

References

Amodei, D., Olah, C., Steinhardt, J., Christiano, P., Schulman, J., and Mane, D. (2016). Concrete Problems in AI Safety. arXiv preprint, 2016.

Chen, L., Zaharia, M., and Zou, J. (2023). FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. arXiv preprint, 2023.

Christiano, P., Leike, J., Brown, T., Martic, M., Legg, S., and Amodei, D. (2017). Deep Reinforcement Learning from Human Preferences. Advances in Neural Information Processing Systems, 2017.

Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Kuttler, H., Lewis, M., Yih, W., Rocktaschel, T., Riedel, S., and Kiela, D. (2020). Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks. Advances in Neural Information Processing Systems, 2020.

Park, J. S., O'Brien, J., Cai, C. J., Morris, M. R., Liang, P., and Bernstein, M. S. (2023). Generative Agents: Interactive Simulacra of Human Behavior. arXiv preprint, 2023.

Schick, T., Dwivedi-Yu, J., Dessi, R., Raileanu, R., Lomeli, M., Zettlemoyer, L., Cancedda, N., and Scialom, T. (2023). Toolformer: Language Models Can Teach Themselves to Use Tools. arXiv preprint, 2023.

Shen, Y., Song, K., Tan, X., Li, D., Lu, W., and Zhuang, Y. (2023). HuggingGPT: Solving AI Tasks with ChatGPT and its Friends in Hugging Face. arXiv preprint, 2023.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., and Wang, C. (2023). AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation. arXiv preprint, 2023.

Yang, J., Jimenez, C. E., Wettig, A., Lieret, K., Yao, S., Narasimhan, K., and Press, O. (2024). SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv preprint, 2024.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. (2023). ReAct: Synergizing Reasoning and Acting in Language Models. International Conference on Learning Representations, 2023.