LUNARZERO: A SINGLE-BINARY CODING AGENT THAT RUNS ON POOLED FREE-TIER LANGUAGE MODELS WITH QUOTA-AWARE FAILOVER

Anonymous authors
Paper under double-blind review

ABSTRACT

Agentic software engineering systems are commonly evaluated under the assumption that their language model is continuously available. This assumption excludes an important deployment setting: users and institutions that cannot afford sustained access to premium APIs or the infrastructure required to self-host capable models. Such users often depend on free or otherwise rate-limited access, where individual models have limited request and token budgets, availability changes over time, and failures may prevent an agent from completing a task even when alternative models remain accessible. We study this setting empirically. We build LUNARZERO, a terminal coding agent that treats models from multiple providers as a shared pool and maintains a persistent per-model ledger of request and token usage, failure-class cooldowns, and pre-streaming failover, and we evaluate it on all 300 tasks of SWE-bench Lite using the official instance images and evaluator. Across 886 evaluator-scored runs the pooled configuration resolves 42.3% of tasks at zero model cost, using 98 distinct models from ten providers. We compare it against a matched control that uses the same agent and protocol pinned to the model ranked highest by the router's quality metric and used most by the pooled system. On the 186 tasks attempted by both, the pooled system resolves 36.6% against 41.9% for the control, a paired difference of -5.4 percentage points (95% CI $[-13.4, +2.7]$; exact McNemar, $p = 0.24$): under the evaluated configuration we find no statistically detectable difference in task resolution. The configurations nevertheless exhibit substantially different failure modes. The pooled system abandons tasks without producing a patch far more frequently ($+28.0$ pp, $p < 10^{-12}$), whereas the control more frequently exhausts its time budget once its provider becomes unavailable (-15.6 pp, $p < 10^{-5}$); the pooled system also completes tasks $1.7\times$ faster and sustains far more total model usage than any individual free tier. Ablations show the hand-designed cooldown policy to be conservative: uniform exponential backoff increases resolution by 18.0 pp on its paired subset ($p < 10^{-3}$) at roughly $6\times$ the request volume. We characterize 47,406 failover events, of which over a quarter arise from catalogue drift rather than rate limiting. These results characterize model availability as a distinct systems constraint: pooling rate-limited models can substantially increase aggregate availability and throughput without necessarily improving task resolution. We specify the routing policy and evaluation protocol in full, and report the accounting behind every claim, so that the study can be replicated independently.

1 INTRODUCTION

Agentic software engineering systems use language models in iterative loops involving code search, file modification, execution, and feedback. Recent systems have demonstrated substantial capability on real-world software engineering benchmarks such as SWE-bench (Jimenez et al., 2024), but their evaluations typically assume that the configured model is continuously available throughout an agent's execution.

That assumption is natural for paid APIs and self-hosted models, but it is restrictive for users relying on free or rate-limited access. A free model may impose request-per-minute, request-per-day,

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.

token-per-minute, or token-per-day limits, and providers return heterogeneous errors when limits are reached, models are retired, access tiers change, or requests violate provider-specific constraints. An agent can therefore encounter an unavailable model even when other models remain accessible.

This setting is particularly relevant to researchers, educators, learners, independent developers, and institutions with limited computational and financial resources. For these users, continuously available premium APIs or sufficiently powerful self-hosted models may be prohibitively expensive or operationally infeasible. Free-tier access can broaden participation in agentic software engineering, but its fragmented quotas and changing availability make reliable execution difficult.

This creates a systems problem that differs from conventional model routing. In standard routing and cascading settings the principal question is which model should process a request; under exhaustible access the agent must additionally decide whether a model is currently available, whether it is worth waiting for that model to recover, and whether another should be used instead. Availability is dynamic, partially observed, and coupled across tasks through shared quotas. A natural response is to pool multiple models and fail over when the current one becomes unavailable, but pooling introduces its own trade-off: switching away from a high-capability model may reduce the probability that the next model completes the task, while refusing to switch may leave the agent idle or exhaust its time budget. The effect on end-to-end performance is therefore an empirical question.

We study this question with LUNARZERO, a terminal coding agent that represents multiple free-tier models as a single logical pool Its router is a fixed, hand-engineered policy rather than a learned routing method, and its parameters were frozen before benchmark evaluation. We evaluate it on all 300 SWE-bench Lite test instances in the official benchmark environment, scored by the official evaluator. To isolate the effect of pooling from differences in the underlying agent, we compare the pooled configuration against a matched control that uses the same agent and protocol while pinned to a single model, defined operationally rather than as an independently established strongest model

The resulting picture is more nuanced than a simple accuracy comparison. Pooling produces no statistically detectable improvement in resolution on the matched subset; instead the two configurations fail in different ways. The pooled system switches away from unavailable models and continues making progress across a larger pool, but sometimes hands work to models that terminate without producing a patch, whereas the control preserves a stable capability profile and becomes increasingly constrained as its provider's quota is exhausted. The pooled configuration therefore processes tasks more quickly and sustains greater aggregate model usage at statistically indistinguishable resolution.

The study also reveals a methodological issue. When experiments share exhaustible quotas, aggregate comparisons across different task subsets can confound an algorithmic change with the state of the shared resource. We initially observed an apparent negative effect from removing the availability ledger; pairing each ablation with the reference configuration on the same instances eliminates it. This motivates paired evaluation and explicit quota-state control as requirements for benchmarking under shared rate limits. In sum, our contributions are as follows.

i. **Resource-constrained agentic computing**, demonstrating the feasibility and the limitations of running a large-scale agentic software engineering evaluation without model-access charges, and providing an empirical foundation for more accessible agentic systems aimed at users who cannot rely on continuously available premium models.

ii. **An evaluation protocol** for exhaustible model access, accounting for shared request and token quotas, fresh quota state, fixed task ordering, per-task evaluation, and paired comparisons.

iii. **An empirical evaluation of model pooling** for agentic software engineering: across 886 evaluator-scored runs on SWE-bench Lite we characterize task resolution, throughput, failure composition, model usage, and quota exhaustion under pooled and single-model configurations.

iv. **A failure taxonomy for multi-provider model access**, distinguishing rate limiting, catalogue drift, authentication failures, and request-shape incompatibilities among 47,406 failover events.

v. **An analysis of routing-policy trade-offs**, showing the hand-designed cooldown policy to be conservative while removing availability accounting produces no detectable resolution change.

The resulting study does not necessarily establish that model pooling improves software engineering accuracy. Instead, it shows that availability and task resolution can diverge in agentic systems, and provides an empirical baseline for future policies that model the decision to wait versus switch.

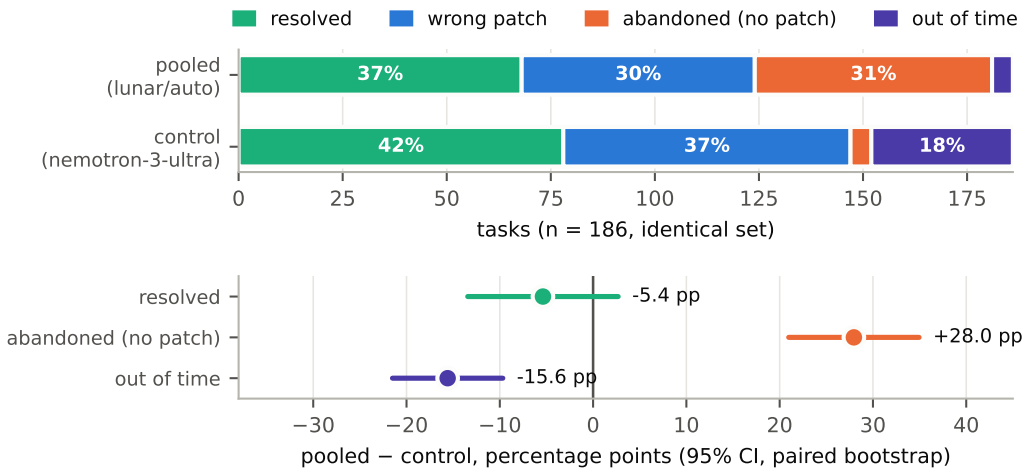

Figure 1: Outcome composition on the 186 SWE-bench Lite tasks attempted by both configurations, with 95% paired-bootstrap intervals. The difference in resolution is not statistically detectable ($p = 0.24$), whereas the differences in abandonment and time-budget exhaustion are ($p < 10^{-5}$). The two configurations attain comparable resolution through different failure profiles.

2 Related Work

Language-model-based software engineering agents combine iterative inference with tools for repository inspection, code modification, and execution. ReAct (Yao et al., 2023) established a general framework for interleaving reasoning and action, while SWE-agent (Yang et al., 2024) demonstrated the importance of the agent–computer interface for software engineering. Subsequent systems, including OpenHands, AutoCodeRover, Agentless, and MAGIS, explore different approaches to repository navigation and issue resolution (Wang et al., 2025; Zhang et al., 2024; Xia et al., 2024; Tao et al., 2024). Recent benchmarks have expanded agent evaluation to realistic, interactive, and execution-based environments (Liu et al., 2024; Zhou et al., 2024; Ma et al., 2024; Xie et al., 2024; Jain et al., 2025; Badertdinov et al., 2025). These studies primarily evaluate agent capability under configured model access; our focus is complementary, concerning how changing model availability affects execution when access is rate-limited and exhaustible.

Existing routing and cascading methods allocate requests among models according to predicted quality, query difficulty, or inference cost (Chen et al., 2024; Madaan et al., 2024; Ding et al., 2024; Ong et al., 2025; Dekoninck et al., 2025; Jitkrittum et al., 2026). These methods generally treat model availability as exogenous: once selected, a model is assumed to be able to serve the request. In our setting availability is instead a dynamic, partially observed state, because request or token quotas can be exhausted, providers can throttle access, and advertised model catalogues can change. SWE-Effi (Fan et al., 2025) evaluates software-engineering agents under token, time, and monetary-cost constraints. Our study addresses a related but distinct problem by treating quotas as shared, exhaustible state that can make a model unavailable during an agent's execution. A more comprehensive discussion is provided in Appendix A.

3 System Design

Figure 2 summarises the study. LunarZero is a Rust workspace of five crates (approximately 41k lines, 127 tests) that compiles to a single static binary with no runtime service dependency. The agent loop is conventional: one provider call per step, concurrent tool execution, and a history that is re-read at every step. Because each request is rebuilt per step, a step may be retried on a different model. Model access uses Anthropic's Messages API for Claude models and OpenAI-style chat completions otherwise, including for all free-tier providers. We describe only the components that the experiments manipulate; the token-economy and permission layers are in Appendices B and C.

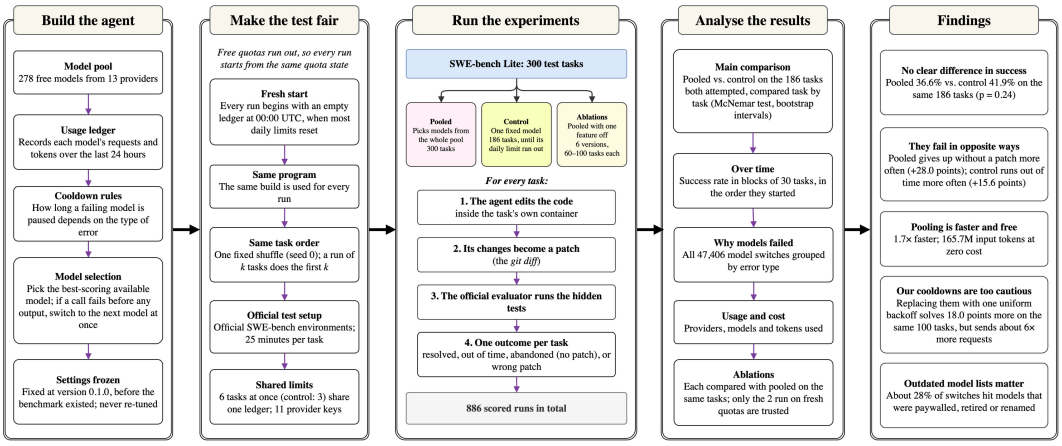

Figure 2: Overview of the study design, experiments, analysis, and principal findings in a five-stage workflow: from building and freezing the LUNARZERO agent and establishing fair test conditions, to running the evaluations, analysing outcomes, and summarising the main findings on success, failure modes, speed, cost, cooldowns, and model-list freshness.

Table 1: Cooldown classes derived from failure responses, where n denotes the number of consecutive failures of the model and a success resets n. Section 6 shows that this policy is measurably conservative.

Signal	Scope	Cooldown
429 with `retry-after`	model	$\max(\text{ra}, 2\,\text{s}) + 1\,\text{s}$
429, daily-quota wording	model	until UTC midnight
429, other	model	$60\,\text{s} \cdot 2^{\min(n,6)}$, $\leq$1 h
401/403, key invalid	provider	1 h
401/403, model-scoped wording; 402; 404/410	model	24 h
413 or "too large"	model	3 min
5xx, network, timeout	model	$30\,\text{s} \cdot 2^{n}$, $\leq$10–15 min

Pool and catalogue. A build script derives a pool of 278 models from a snapshot of the models.dev catalogue by applying thirteen providers' free-tier rules and published caps. Each model carries a quality score in $[0, 100]$ derived from parameter count, the price the same weights command on paid providers, release date, reasoning support, and context window, together with a speed prior from provider throughput. The catalogue is compiled into the binary.

Availability ledger. A persistent ledger records, per model, request timestamps and token counts over the preceding 24 hours, a cooldown deadline, a consecutive-failure count, the most recent error, and moving averages of time-to-first-token and throughput, together with a provider-wide cooldown. A model is *available* for a request of estimated size $\hat{T}$ if no cooldown is active and no window would be exhausted by the request; its headroom is $h = \min_w (1 - \text{used}_w/\text{cap}_w)$ over the RPM, RPD, TPM, and TPD windows. Section 6 ablates this component.

Failure-class cooldowns. Published caps are approximate and provider errors are heterogeneous, so the cooldown duration is derived from the *class* of the failure (Table 1). A `retry-after` header is honoured exactly rather than fed into a backoff schedule; daily-quota wording suspends the model until the next UTC midnight; an invalid key suspends the *provider* for one hour, whereas a model-scoped message suspends only that model, for one day. Section 6 ablates this component.

Selection and failover. Each request carries a *need*: the required capabilities, the user's text, and an estimate $\hat{T} = 1.15 \cdot \text{chars}/4 + 2{,}600$ of prompt tokens. Candidates are connected pool models that satisfy the capability requirements and whose context window is at least $1.1\hat{T} + 2{,}048$. A scored classifier assigns the text to one of *chat*, *coding*, *reasoning*, or *long-context*, selecting weights

(w_q, w_s, w_h); each available candidate is scored $S = w_q\, q/100 + w_s\, s + w_h\, h - 0.05\, n + \nu + \lambda + \pi$, where s combines the speed prior with measured latency, n is the consecutive-failure count, and ν, λ, π are task, latency, and policy adjustments. Sticky routing retains a session's model for 30 minutes unless an available model exceeds it by 12 quality points. On a failure *before any output has streamed* the router records it, selects the next untried candidate, and rebuilds the request without backoff; on a failure after streaming but before any tool call has executed, the partial step is discarded and re-executed. If no model is available, the step waits for the model expected to recover soonest, at most 120 s and three times per step.

Hyperparameter provenance. The weights, thresholds, and cooldown constants above were selected during development on interactive use, before any benchmark existed, since an evaluation of a hand-designed policy is uninformative if the policy was tuned on the evaluation. In our repository the scoring weights, quality formula, cooldown table, and sticky-routing thresholds were last modified in the commit tagging version 0.1.0, which precedes the first commit of the benchmark harness; the only subsequent modification to the routing module adds the switches that disable components for the ablations. No measurement reported here set any of these values, and none were re-tuned after the results were observed. The router should therefore be read as one reasonable hand-designed policy rather than a tuned upper bound, consistent with Section 6.

4 Experimental Setup

Benchmark and harness. We evaluate on all 300 test instances of SWE-bench Lite (Jimenez et al., 2024). Each task executes inside its official `swebench/sweb.eval.*` instance image, the repository at the base commit with its environment installed, with a static Linux build of the agent bind-mounted in. The agent receives the issue text and is instructed to make a minimal source change, not to modify tests, and to terminate when finished; the resulting `git diff` is the model patch, scored by the official evaluator while the image remains local. Six tasks execute concurrently (three in the control), with a 25-minute limit per task; image pulls occur outside that limit.

Protocol for exhaustible quotas. Comparing configurations under free tiers requires controlling for quota state, which is shared, time-varying, and exhaustible. Every configuration begins with a *fresh ledger* at **00:00 UTC**, when most providers' daily windows reset, uses the *same binary*, and follows a fixed task shuffle (seed 0), so a configuration restricted to k tasks executes a uniform random subset of the 300, always the same subset and always at the start of its run. Eleven provider keys were connected. This protocol renders the comparisons below meaningful, and its violation invalidates four of our six mechanism ablations (Section 6).

Configurations. **Pooled**: the full router (`lunar/auto`), 300 tasks. **Control**: the same agent and protocol pinned to `nemotron-3-ultra-550b`. The operational definition rests on two criteria that agree: the router's quality metric ranks it highest among connected models, and the pooled configuration used it most, serving 71.4% of its steps. It completed 186 tasks before its provider's daily budget was exhausted, which is itself an observation. **Ablations**: the pooled configuration with a single mechanism disabled, 60–100 tasks each, compared against it *on the same instances*.

Statistical analysis. Task outcomes are binary and paired, since the same instance is attempted by both configurations. We therefore report exact McNemar tests on the discordant pairs and 95% paired-bootstrap intervals ($20{,}000$ resamples over tasks) for every difference, together with marginal bootstrap intervals for individual rates. One caveat applies throughout: tasks within a configuration share a single quota ledger, so they are not independent draws, and the reported intervals therefore *understate* uncertainty. This is a property of the setting rather than of the estimator, since any evaluation conducted against shared exhaustible quota exhibits it, and it is a further reason to treat the resolution comparison as inconclusive rather than as evidence favouring the control.

Outcome definitions. We report the official evaluator's verdict and decompose non-resolution into *wrong patch*, *abandoned* (the agent terminated its turn without producing a patch), and *out of time* (the 25-minute limit), since these separate the configurations far more sharply than resolution does. The four categories form a *partition*, applied in the order resolved, out of time, abandoned,

Table 2: SWE-bench Lite under the official evaluator, at zero model cost. Intervals are 95% bootstrap over tasks; "Failovers" counts mid-step model switches, not per-step selections. Ablation rows are *not* comparable to the pooled row's 42.3%, since each executed a different task subset; the paired comparisons appear in Table 3.

Configuration	n	Resolved (95% CI)	Wrong	Aband.	Timeout	Steps	Failov.
pooled	300	42.3% [36.7, 48.0]	26.3%	24.7%	6.7%	24	35.4
matched subset	186	36.6% [29.6, 43.5]	30.1%	30.6%	2.7%	25	16.7
control	186	41.9% [34.9, 48.9]	37.1%	2.7%	18.3%	41	32.9
no cooldown classes	100	45.0%	36.0%	7.0%	12.0%	26	204.5
no ledger	60	28.3%	38.3%	30.0%	3.3%	18	51.2

wrong patch, so a task reaching the limit having produced nothing is counted once. Treating them as independent indicators would double-count the five tasks in the matched subset satisfying two.

5 Results

5.1 Matched Comparison of Resolution and Failure Composition

Our primary analysis concerns the 186 tasks attempted by both configurations. The contingency table is: both resolved 44, pooled only 24, control only 34, neither 84, giving $68/186 = 36.6\%$ for the pooled configuration and $78/186 = 41.9\%$ for the control. The paired difference is -5.4 percentage points (CI $[-13.4, +2.7]$), and an exact McNemar test on the 58 discordant pairs yields $p = 0.24$. The interval contains zero and the test does not reject; at this sample size the configurations cannot be distinguished on resolution. We therefore do not describe the control as superior. The supportable claim is weaker: under the evaluated configuration and routing policy, pooling did not improve resolution relative to a single connected model.

The differences that are statistically detectable concern failure composition. The pooled configuration abandoned 30.6% of tasks without producing a patch, against 2.7% for the control, a paired difference of $+28.0$ pp (CI $[+21.0, +34.9]$, $p = 9.1 \times 10^{-13}$). The control exhausted its time budget on 18.3% of tasks against 2.7%, a difference of -15.6 pp (CI $[-21.5, -9.7]$, $p = 1.1 \times 10^{-6}$). The remaining category, a patch that fails the hidden tests, does not separate the configurations (-7.0 pp, CI $[-15.6, +1.6]$, $p = 0.14$): conditional on producing a patch at all, either configuration is approximately equally likely to be correct. The pooled configuration required 7.6 minutes and 25.5 steps per task, against 13.1 minutes and 48.7 steps for the control, a $1.7\times$ throughput advantage. Two configurations with statistically indistinguishable accuracy thus occupy opposite regions of the failure space, and this dissociation, rather than a ranking on resolution, is the principal finding.

The mechanism is visible in the execution traces. The pooled router transfers a task to a weaker model at the first rate-limit signal, and weaker models frequently terminate the turn by *describing* the intended change rather than invoking an edit tool; 27 pooled tasks terminated after a single assistant step in this manner, none of which resolved. The control never switches and never inherits another model's behaviour, but cannot escape its provider's limits.

Figure 3 shows the temporal structure that the aggregate conceals, and the two configurations move in opposite directions. Over its first 120 tasks, on fresh quota, the control resolved 55.0%. Over its final 60, after its provider began returning 429 and tier-scoped 401 responses, it resolved 15.0%, and its last block resolved none. The pooled configuration exhibits no such decay; it instead improves, from 30.0% over its first 120 started tasks to 51.7% over its last 60.

Cold-start behaviour of the pooled router. We report this as an observation with a hypothesis rather than a demonstrated mechanism. With an empty ledger the router has no measured latency and no failure history, so concurrent tasks select the same highest-scoring model, exhaust its per-minute window simultaneously, and cascade together into weaker models that abandon the task; as the ledger accumulates observations, load disperses. This predicts that the effect tracks ledger state rather than elapsed time, which is testable by warming the ledger before a run, an experiment we

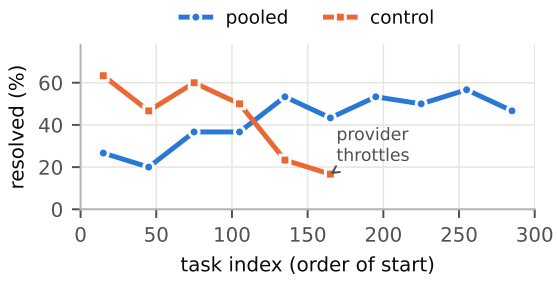

Figure 3: Resolution rate in blocks of 30 tasks in start order. Ordering by completion time would place fast failures first and produce a spurious trend. The control begins high on fresh quota and declines to zero as its provider returns 429 and tier-scoped 401 responses; the pooled configuration improves from a cold start without collapsing.

have not performed. The pooled 42.3% therefore combines a cold-start with a steady-state regime, and any configuration executing only a prefix of the task order oversamples the former (Section 6).

5.2 Provider and Quota Accounting

Appendix D gives the exact accounting. Our catalogue contains thirteen providers, eleven had keys connected, and ten served at least one step; the remaining connected provider, `github-models`, was preempted at every selection. The pooled configuration issued $8{,}870$ steps across 98 models, with a highly concentrated distribution: one provider served 71.4% of steps and the next two 16.3%. This concentration is what makes the control meaningful, since it pins the agent to the model the pooled configuration already relied on, and is equally a constraint on the generality of the result.

5.3 Token Volume and Monetary Cost

The pooled configuration consumed 165.7M input and 1.3M output tokens at zero cost. The largest published daily token cap among these providers is 1M, so no single free model could have served more than a small fraction of that volume; the control's exhaustion after 186 tasks is the same constraint observed directly. Pooling is what renders an evaluation of this scale feasible. This is the clearest positive result in this study, and it concerns availability rather than accuracy.

5.4 Taxonomy of Failover Events

Across all configurations the router recorded **47,406** failovers; the complete breakdown appears in Appendix E. Rate limiting dominates (55.6%), but 27.8% of all failovers arise from *catalogue drift*: models that the published catalogue lists as freely available but that are paywalled at run time (402, 14.9%) or retired or renamed ($404/410$, 12.9%). Authentication failures (6.8%) comprise both expired keys and tier-scoped refusals, which must be attributed to a model rather than to a provider, and request-shape rejections (5.0%) reflect provider-specific validator disagreements.

Separating catalogue drift from rate limiting. The two mechanisms admit different remedies, since drift is addressable by a more accurate catalogue and rate limiting is not, so we separate them. Within the pooled configuration, tasks encountering at least one drift failover resolved at 40.3% ($n = 77$) against 43.0% for those encountering none ($n = 223$), a difference small relative to the interval on either rate. Drift consumes failover attempts and request budget without materially affecting resolution: the router identifies the unavailable model, applies a cooldown, and proceeds within the same step. Rate limiting reaches the outcome, because it removes models the router selected rather than models it can immediately replace. The contrast in which tasks with no rate-limit failover ($n = 35$) resolved at 8.6%, is selected on the outcome, since tasks that never encounter a rate limit are mostly those that terminated early for other reasons, and we do not interpret it.

Table 3: Ablations, each compared against the pooled configuration *on that ablation's own instances*. Intervals are 95% paired bootstrap and p is exact McNemar. Only the first two rows satisfy the fresh-quota protocol of Section 4; the remaining four executed consecutively against a single exhausted pool and are reported only to illustrate the consequences of a protocol violation. Re-runs are in progress.

Ablation	n	Ablated	Pooled (same n)	Difference (95% CI)	p
Fresh pool: protocol satisfied					
no cooldown classes	100	45.0%	27.0%	$+18.0$ $[+8.0, +28.0]$	9×10^{-4}
no ledger	60	28.3%	23.3%	$+5.0$ $[-10.0, +20.0]$	0.66
Exhausted pool: not interpretable as mechanism effects					
no plan continuation	60	1.7%	23.3%	-21.7 $[-31.7, -11.7]$	2×10^{-4}
no sticky routing	60	8.3%	23.3%	-15.0 $[-26.7, -3.3]$	0.02
no write stubbing	60	8.3%	23.3%	-15.0 $[-26.7, -3.3]$	0.02
no self-healing	60	11.7%	23.3%	-11.7 $[-23.3, +0.0]$	0.09

Defects surfaced only at scale. Two defects in cross-family failover appeared only at this scale, both since corrected. Gemini 3 rejects a history whose tool calls were produced by another model family (HTTP 400, missing thought signature), cascading through every Gemini model in sequence; the correction supplies the documented placeholder for migrated traces. Following that correction the passthrough field was replayed to all providers, and Mistral's validator rejects the unknown field (HTTP 422), suspending all 30 Mistral models for 30 minutes after any Gemini step. The reported 42.3% predates the second correction and is therefore a lower bound.

6 Ablation Studies

The necessity of paired ablation comparisons. Each ablation executed a random subset of the 300 instances, so its resolution rate is not comparable to the pooled configuration's 42.3% over all 300. Read against the full-run aggregate, removing the ledger appears to cost 14 points (42.3% $\rightarrow$ 28.3%), a conclusion we initially accepted. Against the pooled configuration *on the same 60 instances*, where it resolved 23.3%, the ablation yields $+5.0$ pp (CI $[-10.0, +20.0]$, $p = 0.66$): no detectable effect, with the point estimate changing sign. We present this correction because the same error is easily made in any evaluation whose arms run at different scales against shared quota.

Conservatism of the failure-class cooldowns. Replacing the failure-class cooldowns with a single uniform exponential backoff, essentially the gateway policy of BerriAI (2023), increases the resolution rate by 18.0 pp on the same instances (CI $[+8.0, +28.0]$, $p = 9 \times 10^{-4}$) and reduces abandonment from 27% to 7%, at 204.5 failovers per task rather than 35.4. Suspending a model for minutes to hours following a 429 keeps the strongest models unavailable for the rest of a task, whereas retrying them without discrimination recovers accuracy at about six times the request volume, with 12% of tasks still reaching the time limit. This is the clearest mechanism result in the study, and a negative one about our own design: the policy frozen before the evaluation (Section 3) errs on the conservative side. It is not a recommendation to remove cooldowns, since a sixfold increase in requests against published caps is a different trade-off rather than a costless improvement.

Four ablations confounded by quota exhaustion. The self-healing, sticky-routing, write-stubbing, and plan-continuation configurations executed consecutively against a single day's quota, scoring 1.7–11.7% with median 1–6 steps. These figures measure pool exhaustion rather than the mechanism removed. Their paired differences in Table 3 are large and several nominally significant, which is the point: a significance test does not rescue a confounded comparison. Under rate-limited access, an experiment that does not control quota state measures the quota. We exclude them from every claim; they are being re-run individually on fresh quota at $n = 60$, which is the largest size the pool sustains within a single daily window.

Cold-start bias in prefix-limited configurations. The pooled configuration resolved 23.3% on the 60 ablation instances and 47.1% on the remaining 240, although the subset is a uniform ran-

dom draw with proportionally identical repository composition. The explanation is the cold start of Section 5.1: a configuration limited to k tasks executes the *first* k of the shuffled order, which the pooled configuration attempted during its own cold start, resolving 26.7% and 20.0% in its first two blocks of 30. The pairing is sound, since both encountered these instances with an empty ledger, but the comparisons in Table 3 are cold-start comparisons and need not hold in steady state. With the interval widths at $n = 60$, this motivates both the pending re-runs on fresh quota and a second seed.

7 DISCUSSION

The wait-versus-switch decision. Our three fresh-pool configurations bracket a decision the agent makes thousands of times per run: switching immediately trades patches for time, never switching is accurate while quota remains and unproductive afterwards, and retrying without discrimination recovers accuracy at six times the request volume. Our measurements do not identify an optimal policy. They establish that the three differ significantly in failure composition while two are statistically indistinguishable in resolution, suggesting that the quantity worth optimising here is not expected quality per request. The signals a better policy would require are recorded: per-model latency, observed `retry-after` values, window occupancy, the failure class, and the fact that a rate limit is a statement about *time*, so waiting carries a computable cost. Formulating this as sequential decision-making over rate-limited arms under censored feedback, in which a 429 reveals that a cap was reached but not where it lies, appears natural, but is a direction rather than a result.

Compliance, capability, and generality. LUNARZERO uses a single user-supplied key per provider and its ledger exists to remain *within* published caps; it does not rotate keys or retry a model that has refused. We note the tension with the no-cooldown ablation: the configuration scoring highest issues the most requests, and we do not recommend it. Pooling also cannot manufacture capability, and here it did not improve upon a single connected model's accuracy, although it did render an evaluation of this scale feasible. Nothing in the ledger, cooldown classes, or failover mechanism is specific to software engineering, but we have not evaluated that generality.

Limitations. The findings should be interpreted in light of the single task ordering of the study, the shared quota state, the choice of control, and the time-specific snapshot of free-tier availability. A detailed discussion is provided in Appendix K.

8 CONCLUSION

Free model tiers make agentic software engineering feasible without a budget, and pooling them makes an evaluation of this scale feasible at all: 165.7M tokens, 98 models, ten providers, at zero cost. On accuracy, neither the intuitive expectation nor its converse survives measurement. A pooled router and a single connected model resolve statistically indistinguishable fractions of SWE-bench Lite while failing in opposite and clearly distinguishable ways: under the evaluated configuration, pooling purchased availability and throughput but not resolution. Our ablations, once paired correctly, retract one of our own conclusions and sharpen another, the cooldown policy being measurably conservative. This study takes a step toward making agentic software engineering accessible beyond organizations with substantial API or computing budgets. Our results show that pooling free-tier models can provide considerable aggregate availability and throughput at zero model cost, although access alone does not guarantee reliable task completion. Democratizing agentic systems therefore requires not only affordable model access, but also routing policies that manage changing availability without sacrificing too much capability.

AI USE STATEMENT

In this work, we used generative AI tools (Claude, Anthropic) for preparing an initial version of the methodology overview diagram (Figure 2) based on our study information and instructions. The authors subsequently reviewed, substantially revised, corrected, and finalized the figure to ensure its accuracy and consistency with the methodology and results. The system design, the experimental protocol, all measurements reported in Section 5 and the interpretation of the results were carried out by the authors. Additionally, we used generative AI tools to edit text, tightening wording to

meet the page limit without changing any claim or number. We have reviewed all AI-assisted work: the methodology diagram and every edited sentence were checked against the reported results, and all code was reviewed and tested (127 tests, continuous integration on three operating systems). We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

LUNARZERO uses a single user-supplied key per provider, operates within published limits, and does not rotate keys or circumvent quotas (Section 7); agent-initiated installation of third-party code requires human confirmation (Appendix C). No human-subject study was conducted.

REPRODUCIBILITY STATEMENT

Neither the agent, the harness, nor the per-task run records are released. We have therefore aimed to make the study replicable from its specification alone, and to report results at a granularity that leaves no quantity unstated. The paper gives the routing policy and every constant it depends on (Section 3 and Table 1), the hyperparameter provenance (Section 3), the evaluation protocol for exhaustible quotas (Section 4), the harness including image handling, isolation, patch extraction and scoring with the official SWE-bench evaluator (Appendix I), the ablation switches and the module each one disables (Appendix H), and the statistical procedures with the exact test statistic and resampling scheme (Appendix F). Results are reported as complete aggregates rather than selected summaries: outcome composition for every configuration (Table 2), every paired ablation comparison with intervals and p-values (Table 3), the full failover taxonomy with counts for every class (Table 5), per-provider step counts and published caps (Table 4), and per-repository resolution (Table 6). A replication is expected to differ in absolute numbers, since free-tier terms change without notice; the protocol of Section 4 is what makes the comparisons within a run meaningful, and it is the comparisons, not the absolute rates, that carry the paper's claims.

REFERENCES

Anthropic. Claude Code, 2025. URL https://docs.anthropic.com/en/docs/claude-code.

Ibragim Badertdinov, Alexander Golubev, Maksim Nekrashevich, Anton Shevtsov, Simon Karasik, Andrei Andriushchenko, Maria Trofimova, Daria Litvintseva, and Boris Yangel. SWE-rebench: An automated pipeline for task collection and decontaminated evaluation of software engineering agents. In *Advances in Neural Information Processing Systems*, volume 38, 2025. doi: 10.52202/085713-0788. URL https://proceedings.neurips.cc/paper_files/paper/2025/hash/21bec6ace947b1b58967b945c8ac0f10-Abstract-Datasets_and_Benchmarks_Track.html. Datasets and Benchmarks Track.

BerriAI. LiteLLM: Call 100+ LLM APIs using the OpenAI format. https://github.com/BerriAI/litellm, 2023.

Lingjiao Chen, Matei Zaharia, and James Zou. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance. *Transactions on Machine Learning Research*, 2024.

Edoardo Debenedetti, Jie Zhang, Mislav Balunović, Luca Beurer-Kellner, Marc Fischer, and Florian Tramèr. AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents. In *Advances in Neural Information Processing Systems 37 (NeurIPS), Datasets and Benchmarks Track*, 2024.

Jasper Dekoninck, Maximilian Baader, and Martin Vechev. A unified approach to routing and cascading for LLMs. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pp. 12987–13010. PMLR, 2025. URL https://proceedings.mlr.press/v267/dekoninck25a.html.

Dujian Ding, Ankur Mallick, Chi Wang, Robert Sim, Subhabrata Mukherjee, Victor Rühle, Laks V. S. Lakshmanan, and Ahmed Hassan Awadallah. Hybrid LLM: Cost-Efficient and Quality-Aware Query Routing. In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024.

Zhiyu Fan, Kirill Vasilevski, Dayi Lin, Boyuan Chen, Yihao Chen, Zhiqing Zhong, Jie M. Zhang, Pinjia He, and Ahmed E. Hassan. SWE-Effi: Re-evaluating software AI agent system effectiveness under resource constraints. *arXiv preprint arXiv:2509.09853*, 2025. URL https://arxiv.org/abs/2509.09853.

Paul Gauthier. Aider: AI pair programming in your terminal, 2024. URL https://github.com/Aider-AI/aider.

Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.

Qitian Jason Hu, Jacob Bieker, Xiuyu Li, Nan Jiang, Benjamin Keigwin, Gaurav Ranganath, Kurt Keutzer, and Shriyash Kaustubh Upadhyay. RouterBench: A Benchmark for Multi-LLM Routing System. In *ICML 2024 Workshop on Foundation Models in the Wild*, 2024.

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. In *Proceedings of the Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=chfJJYC3iL.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. LLMLingua: Compressing Prompts for Accelerated Inference of Large Language Models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik Narasimhan. SWE-bench: Can Language Models Resolve Real-World GitHub Issues? In *Proceedings of the Twelfth International Conference on Learning Representations (ICLR)*, 2024.

Wittawat Jitkrittum, Harikrishna Narasimhan, Ankit Singh Rawat, Jeevesh Juneja, Congchao Wang, Zifeng Wang, Alec Go, Chen-Yu Lee, Pradeep Shenoy, Rina Panigrahy, Aditya Krishna Menon, and Sanjiv Kumar. Universal model routing for efficient LLM inference. In *Proceedings of the Fourteenth International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=ka82fvJ5f1.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, Shudan Zhang, Xiang Deng, Aohan Zeng, Zhengxiao Du, Chenhui Zhang, Sheng Shen, Tianjun Zhang, Yu Su, Huan Sun, Minlie Huang, Yuxiao Dong, and Jie Tang. AgentBench: Evaluating LLMs as agents. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=zAdUB0aCTQ.

Chang Ma, Junlei Zhang, Zhihao Zhu, Cheng Yang, Yujiu Yang, Yaohui Jin, Zhenzhong Lan, Lingpeng Kong, and Junxian He. AgentBoard: An analytical evaluation board of multi-turn LLM agents. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-2365. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/877b40688e330a0e2a3fc24084208dfa-Abstract-Datasets_and_Benchmarks_Track.html. Datasets and Benchmarks Track.

Aman Madaan, Pranjal Aggarwal, Ankit Anand, Srividya Pranavi Potharaju, Swaroop Mishra, Pei Zhou, Aditya Gupta, Dheeraj Rajagopal, Karthik Kappaganthu, Yiming Yang, Shyam Upadhyay, Mausam, and Manaal Faruqui. AutoMix: Automatically Mixing Language Models. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024.

Isaac Ong, Amjad Almahairi, Vincent Wu, Wei-Lin Chiang, Tianhao Wu, Joseph E. Gonzalez, M. Waleed Kadous, and Ion Stoica. RouteLLM: Learning to Route LLMs with Preference Data. In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

OpenRouter. OpenRouter: A unified interface for LLMs. `https://openrouter.ai`, 2023.

Charles Packer, Sarah Wooders, Kevin Lin, Vivian Fang, Shishir G. Patil, Ion Stoica, and Joseph E. Gonzalez. MemGPT: Towards LLMs as Operating Systems. *arXiv preprint arXiv:2310.08560*, 2023.

Wei Tao, Yucheng Zhou, Yanlin Wang, Wenqiang Zhang, Hongyu Zhang, and Yu Cheng. MAGIS: LLM-based multi-agent framework for GitHub issue resolution. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1647. URL `https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d1f02132ef51602adf07000ca5b6138-Abstract-Conference.html`.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayuan Pan, Yueqi Song, Bowen Li, Jaskirat Singh, et al. OpenHands: An Open Platform for AI Software Developers as Generalist Agents. In *Proceedings of the Thirteenth International Conference on Learning Representations (ICLR)*, 2025.

Chunqiu Steven Xia, Yinlin Deng, Soren Dunn, and Lingming Zhang. Agentless: Demystifying LLM-based Software Engineering Agents. *arXiv preprint arXiv:2407.01489*, 2024.

Tianbao Xie, Danyang Zhang, Jixuan Chen, Xiaochuan Li, Siheng Zhao, Ruisheng Cao, Toh Jing Hua, Zhoujun Cheng, Dongchan Shin, Fangyu Lei, Yitao Liu, Yiheng Xu, Shuyan Zhou, Silvio Savarese, Caiming Xiong, Victor Zhong, and Tao Yu. OSWorld: Benchmarking multimodal agents for open-ended tasks in real computer environments. In *Advances in Neural Information Processing Systems*, volume 37, 2024. doi: 10.52202/079017-1650. URL `https://proceedings.neurips.cc/paper_files/paper/2024/hash/5d413e48f84dc61244b6be550f1cd8f5-Abstract-Datasets_and_Benchmarks_Track.html`. Datasets and Benchmarks Track.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*, 2024.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing Reasoning and Acting in Language Models. In *Proceedings of the Eleventh International Conference on Learning Representations (ICLR)*, 2023.

Qiusi Zhan, Zhixiang Liang, Zifan Ying, and Daniel Kang. InjecAgent: Benchmarking Indirect Prompt Injections in Tool-Integrated Large Language Model Agents. In *Findings of the Association for Computational Linguistics: ACL 2024*, 2024.

Yuntong Zhang, Haifeng Ruan, Zhiyu Fan, and Abhik Roychoudhury. AutoCodeRover: Autonomous Program Improvement. *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2024.

Shuyan Zhou, Frank F. Xu, Hao Zhu, Xuhui Zhou, Robert Lo, Abishek Sridhar, Xianyi Cheng, Tianyue Ou, Yonatan Bisk, Daniel Fried, Uri Alon, and Graham Neubig. WebArena: A realistic web environment for building autonomous agents. In *Proceedings of the Twelfth International Conference on Learning Representations*, 2024. URL `https://openreview.net/forum?id=oKn9c6ytLx`.

APPENDIX

1 Introduction **1**
2 Related Work **3**
3 System Design **3**
4 Experimental Setup **5**
5 Results **6**
5.1 Matched Comparison of Resolution and Failure Composition 6
5.2 Provider and Quota Accounting 7
5.3 Token Volume and Monetary Cost 7
5.4 Taxonomy of Failover Events 7
6 Ablation Studies **8**
7 Discussion **9**
8 Conclusion **9**
A Extended Related Work **13**
A.1 Coding Agents and Agent Evaluation 13
A.2 Model Routing and Cascading 14
A.3 Resource-Aware Evaluation 14
A.4 Multi-Provider Gateways and Failure Handling 14
A.5 Context Management and Agent Safety 14
B The Token-Economy Layer **15**
C The Permission and Sandboxing Layer **15**
D Per-Provider Accounting **15**
E Complete Failover Taxonomy **16**
F Statistical Procedures **17**
G Per-Repository Outcomes **17**
H Ablation Definitions **18**
I Benchmark Harness Implementation **18**
J Replication **18**
K Limitations **19**

A Extended Related Work

This appendix expands the discussion of Section 2 by reviewing the literature on coding agents and their evaluation, model routing and cascading, resource-aware evaluation, multi-provider infrastructure, context management, and agent safety.

A.1 Coding Agents and Agent Evaluation

Language-model-based software engineering agents operate through repeated cycles of reasoning, tool invocation, observation, and revision. ReAct (Yao et al., 2023) introduced a general framework for interleaving reasoning and action, providing a foundation for many subsequent tool-using agents. SWE-agent (Yang et al., 2024) showed that the design of the agent–computer interface substantially affects performance on repository-level tasks. OpenHands (Wang et al., 2025) provides a general platform for autonomous software development, whereas AutoCodeRover (Zhang et al., 2024) and Agentless (Xia et al., 2024) explore more structured approaches to fault localization, patch generation, and issue resolution. MAGIS (Tao et al., 2024) instead decomposes GitHub issue resolution across multiple specialized agents. Terminal-based assistants such as Claude Code (Anthropic, 2025) and Aider (Gauthier, 2024) apply related interaction patterns in practical workflows.

Evaluation has expanded beyond isolated code generation and static test sets. AgentBench (Liu et al., 2024) evaluates language models as agents across multiple interactive, multi-turn environments. WebArena (Zhou et al., 2024) and OSWorld (Xie et al., 2024) emphasize reproducible, execution-based evaluation in realistic web and computer environments. AgentBoard (Ma et al., 2024) complements final success rates with intermediate progress measures, enabling more diagnostic analysis of multi-turn behaviour. Within code evaluation, LiveCodeBench (Jain et al., 2025) continuously collects new problems and evaluates generation, execution, self-repair, and test-output prediction. SWE-rebench (Badertdinov et al., 2025) develops a scalable pipeline for collecting fresh, interactive software-engineering tasks and mitigating benchmark contamination. These systems and benchmarks primarily investigate what an agent can accomplish given a configured model, and gen-

erally do not examine whether that model remains accessible throughout a long-running trajectory. Our work studies this complementary constraint by measuring how changing availability, quota exhaustion, and provider-side failures affect execution and resolution.

A.2 Model Routing and Cascading

Model routing selects an appropriate model for each query from a set of candidates. FrugalGPT (Chen et al., 2024) uses cascades to reduce inference cost while preserving response quality. Hybrid LLM (Ding et al., 2024) routes queries according to predicted difficulty and desired quality, while AutoMix (Madaan et al., 2024) combines models through confidence-based escalation. RouteLLM (Ong et al., 2025) learns routing decisions from preference data, and RouterBench (Hu et al., 2024) provides a framework for evaluating multi-model routing systems. More recent work formalizes the relationship between the two paradigms: Dekoninck et al. (2025) develop a unified treatment in which routing selects a model before generation whereas cascading invokes increasingly capable models until a satisfactory result is obtained, and Jitkrittum et al. (2026) study routing across heterogeneous and potentially changing model pools under explicit cost budgets.

These approaches optimize quality–cost trade-offs but generally assume that a selected model can serve the request. Under free or rate-limited access that assumption may fail: a model may have exhausted a request or token window, may have been throttled by its provider, or may no longer be available under its advertised access tier. Routing must therefore account not only for capability and cost but also for dynamic availability, so that waiting for a model to recover, switching to a less capable alternative, and preserving scarce quota for later tasks become temporal resource-allocation decisions.

A.3 Resource-Aware Evaluation

Conventional software-agent benchmarks emphasize task success or issue resolution, but practical deployments also depend on the resources required to obtain that outcome. SWE-Effi (Fan et al., 2025) evaluates software-engineering agents using both resolution performance and consumed resources, including tokens, execution time, and monetary cost, revealing that two systems with similar resolution rates may differ substantially in operational efficiency.

Our setting is related but distinct. Resource-aware evaluation typically treats tokens, time, and expenditure as quantities to be minimized or bounded. We instead treat request and token quotas as shared, exhaustible state: consuming quota on one task changes which models are available to later tasks, and a quota-related failure may interrupt an otherwise viable trajectory. Evaluation under these conditions requires explicit quota-state control, persistent model-level accounting, consistent task ordering, and paired comparisons, and it requires distinguishing failures caused by agent capability from failures caused by model access.

A.4 Multi-Provider Gateways and Failure Handling

Multi-provider gateways such as LiteLLM (BerriAI, 2023) and OpenRouter (OpenRouter, 2023) expose models from different providers through a common interface, reducing integration complexity and supporting fallbacks when a request fails. Client libraries also commonly retry rate-limit and server errors using exponential backoff. These mechanisms usually treat an error as a transient request-level event. Under exhaustible access, however, a failure can carry information about the state of a longer-lived resource window: immediately retrying a model whose daily quota is exhausted wastes requests and time, whereas waiting too long after a short-lived limit unnecessarily reduces throughput. LunarZero therefore maintains persistent per-model availability state and applies failure-specific cooldowns, and the ablations of Section 6 quantify how these choices affect resolution, request volume, and failover behaviour.

A.5 Context Management and Agent Safety

Two further areas relate to components of the system that are not the primary objects of our evaluation. First, long-running agents must manage bounded context windows: MemGPT (Packer et al., 2023) treats context management as a form of virtual memory, while LLMLingua (Jiang et al.,

2023) compresses prompts to reduce context length and inference cost. These methods address the amount of information presented to a model rather than the availability of the model itself. Second, tool-using agents introduce security risks because untrusted repository content can influence model behaviour. Indirect prompt injection can cause an agent to follow malicious instructions embedded in external content (Greshake et al., 2023), and InjecAgent (Zhan et al., 2024) and AgentDojo (Debenedetti et al., 2024) provide benchmarks for evaluating such attacks and defenses. These concerns motivate safeguards in the system design, but the present study evaluates availability-aware model pooling rather than context compression or prompt-injection robustness.

B THE TOKEN-ECONOMY LAYER

Because the conversation is re-sent at every step, per-step overhead and stale history dominate the cost of a long task, and under free tiers token windows are frequently the binding constraint. LUNARZERO holds the fixed portion of a step at approximately 2.6k tokens (the system prompt together with eleven tool schemas, measured at $\approx 2{,}577$) through a terse prompt, schemas reduced to essentials, no replay of model reasoning unless a model family requires it, and descriptions of skills and MCP servers only when the prompt is relevant to them. An approximately 80-token project map allows the model to orient without exploratory tool calls.

Beyond 24k tokens of context, stale tool outputs are pruned to stubs, and the *inputs* of `write` and `edit` calls older than three steps are replaced by a stub of the form `<312 lines, 9,842 chars, on disk>`, since the file itself is the source of truth. Near the routed model's context window, a hidden compaction turn summarises the transcript while retaining the recent tail verbatim. A tree-sitter index over eight languages inserts the definitions a prompt names into the system prompt, together with a signatures-only skeleton of the most relevant file.

These mechanisms reduce token consumption but are not evaluated in the main paper. The ablation that would quantify them (`stubbing`, Appendix H) is among those awaiting a fresh-quota re-run.

C THE PERMISSION AND SANDBOXING LAYER

Every command, file edit, and web fetch is evaluated against an ordered ruleset of {permission, pattern, action} triples, with actions *allow*, *ask*, and *deny*, where the last match wins and the default is *ask*. Patterns for `bash` match the command prefix by arity, so that `rm -rf /` matches the rule for `rm` and an "always allow" response never widens to an entire binary.

The agent is able to install skills and MCP servers from a link. The permission layer distinguishes the *provenance* of such a request. If one of the user's three most recent messages names the source, the installation is treated as an ordinary command. If instead the model obtained the source from content, such as a README, a fetched page, or a tool result, the installation is a *forced ask*: it bypasses every allow rule, is presented to the user regardless of the configured permission mode, and is refused outright when no human is attached to the session. This distinction is structural rather than prompt-based, and is intended to close the path from untrusted content to code execution described by Greshake et al. (2023).

Installer builds (`npm install`, `uv sync`, `cargo build`, `go build`) execute with a scrubbed environment inside `sandbox-exec` on macOS or `bwrap` on Linux, denying reads of `~/.ssh`, `~/.aws`, system keychains, and the tool's own credential file. Both sandboxes were verified to render the credential file unreadable to the build while ordinary work proceeds. Credentials are stored in a mode-0600 file or in the operating system keychain.

D PER-PROVIDER ACCOUNTING

Table 4 reports the exact provider and model accounting for the pooled configuration. The catalogue contains thirteen providers. Eleven had keys connected during the runs, and ten of those served at least one step. The remaining connected provider, `github-models`, was preempted at every selection and served no step; the two catalogued providers without keys are excluded from the table.

Table 4: Per-provider accounting for the pooled configuration (300 tasks, 8,870 model steps, 98 distinct models). "Models used" counts distinct models that served at least one step. Caps are the providers' published free-tier limits at the time of the runs, where RPM, RPD, TPM, and TPD denote requests and tokens per minute and per day; a dash indicates that the provider publishes no limit of that kind.

Provider	Models used	Steps	Share	Published caps (RPM / RPD / TPM / TPD)
kilo	17	6,334	71.4%	20 / 200 / – / –
zai	3	751	8.5%	30 / – / – / 1M
mistral	24	688	7.8%	60 / – / 500k / –
cohere	5	441	5.0%	20 / 33 / – / –
ollama-cloud	16	283	3.2%	20 / 300 / – / –
google	10	203	2.3%	10 / 250 / 250k / –
openrouter	12	91	1.0%	20 / 50 / – / –
groq	5	64	0.7%	30 / 1,000 / 12k / 500k
cloudflare-workers-ai	4	9	0.1%	30 / – / – / –
cerebras	2	6	0.1%	30 / – / 60k / 1M
github-models	0	0	0.0%	15 / 150 / – / –
Total	**98**	**8,870**	**100%**	10 providers served steps

Two properties of this distribution bear on the interpretation of the main results. First, the distribution is highly concentrated: a single provider served 71.4% of steps, which is what makes the single-model control a meaningful comparison, since it pins the agent to the model on which the pooled configuration already relied. Second, the largest published daily token cap among these providers is 1M, whereas the pooled configuration consumed 165.7M input tokens, so no individual free tier could have served the evaluation.

E Complete Failover Taxonomy

Table 5 reports all 47,406 failover events recorded across every configuration, classified by the response that triggered them. A failover is a mid-step transition from one model to another, and is distinct from the per-step model selection performed at the start of each step.

Table 5: Failover events by class, across the 886 evaluated tasks. Catalogue drift comprises the 402 and 404/410 rows, which together account for 27.8% of all failovers and arise from models that the published catalogue lists as freely available but that are paywalled, retired, or renamed at run time. Counts are reported for every class, so the taxonomy is complete rather than selective.

Class	Interpretation	Count	Share
429	rate limited, per-minute or per-day window	26,375	55.6%
402	catalogued as free, paid at run time	7,059	14.9%
404 / 410	model retired or renamed	6,114	12.9%
401 / 403	authentication, expired or tier-scoped key	3,245	6.8%
400 / 422	request shape rejected by provider validator	2,369	5.0%
internal	loop guard, waits and other internal conditions	1,350	2.8%
5xx	provider overloaded or unavailable	629	1.3%
413	request too large for the model	265	0.6%
Total		**47,406**	**100%**

The 400/422 row is dominated by a single defect described in Section 5.4, in which a Google-specific passthrough field was replayed to all providers and rejected by Mistral's validator. The 401/403 row conflates two conditions that require different responses, namely an invalid key, which should suspend the provider, and a tier-scoped refusal naming a specific model, which should suspend only that model. The router distinguishes them by message wording (Table 1).

F Statistical Procedures

Task outcomes are binary and paired, since the same benchmark instance is attempted by both configurations under comparison. We therefore use the following procedures throughout.

Exact McNemar test. For a paired comparison we form the contingency table over instances and consider only the discordant pairs, of which b are resolved by the first configuration alone and c by the second alone. Under the null hypothesis of no difference, $b \sim \mathrm{Binomial}(b + c, 1/2)$, and we report the exact two-sided binomial probability $p = \min\Big(1,\ 2\sum_{k=0}^{\min(b,c)} \binom{b+c}{k} 2^{-(b+c)}\Big)$. We use the exact form rather than the χ^2 approximation because several comparisons involve fewer than 25 discordant pairs.

Paired bootstrap intervals. For every reported difference we resample instances with replacement $20{,}000$ times, recomputing the difference within each resample so that the pairing is preserved, and report the 2.5th and 97.5th percentiles. Marginal rates are given intervals by the same procedure applied to a single configuration.

Outcome partition. The four outcome categories form a partition, assigned in the order resolved, out of time, abandoned, wrong patch. A task that reaches the time limit having produced no patch is counted once, as out of time, and a task that reaches the limit but nonetheless resolves is counted as resolved. Treating the categories as independent indicators would double-count the five tasks in the matched subset that satisfy two of them.

Dependence between tasks. Tasks within a configuration share a single quota ledger, so they are not independent draws. The intervals above therefore understate uncertainty by an amount we cannot quantify from a single run. This is a property of evaluation under shared exhaustible quota rather than of the estimator, and it is the principal reason a second seed is the most valuable addition to this work.

G Per-Repository Outcomes

Table 6 reports resolution by source repository for the pooled configuration. SWE-bench Lite is dominated by two repositories, which together account for 191 of 300 instances, so repository composition is a material consideration when comparing task subsets. We verified that the 60-instance and 100-instance ablation subsets are proportionally identical in composition to the full set, which is why Section 6 attributes the difference in the pooled configuration's performance on those subsets to cold-start effects rather than to instance difficulty.

Table 6: Resolution by repository for the pooled configuration (300 instances).

Repository	Resolved / attempted	Rate
django	55 / 114	48.2%
sympy	31 / 77	40.3%
scikit-learn	12 / 23	52.2%
matplotlib	6 / 23	26.1%
pytest	5 / 17	29.4%
sphinx	6 / 16	37.5%
requests	2 / 6	33.3%
astropy	4 / 6	66.7%
pylint	4 / 6	66.7%
xarray	0 / 5	0.0%
seaborn	2 / 4	50.0%
flask	0 / 3	0.0%
Total	**127 / 300**	**42.3%**

H Ablation Definitions

Each ablation disables exactly one mechanism through the environment variable `ABLATE`, which is read once per process. All other behaviour, including the model catalogue, the prompt, and the tool set, is identical across configurations, and every configuration uses the same compiled binary. Table 7 lists the switches and the modules they affect.

Table 7: Ablation switches. The first two were evaluated on a fresh pool and are reported in Section 6; the remaining four executed against an exhausted pool and are excluded from all claims pending re-runs.

Flag	Mechanism disabled	Module
`cooldowns`	failure-class cooldowns, replaced by uniform backoff	`provider/router.rs`
`ledger`	request and token window accounting	`provider/router.rs`
`sticky`	sticky sessions; every step re-selects a model	`provider/router.rs`
`stubbing`	stubbing of old write and edit inputs	`session/history.rs`
`heal`	self-healing from diagnostics and failed checks	`session/runner.rs`
`plan`	plan continuation to open task items	`session/runner.rs`

I Benchmark Harness Implementation

Instance images and architecture. Instance images are the official `swebench/sweb.eval.*` images. Of the 300 instances, 52 have native `arm64` variants and the remaining 248 execute under x86-64 emulation on our Apple Silicon host. Image pulls occur before the per-task clock starts, so tasks executed early in a run are not penalised for a cold image cache. The evaluator runs against a locally cached dataset whose image names are pinned to the host architecture, ensuring that the agent and the evaluator use the same image.

Isolation and shared state. Each task receives its own XDG data directory, because the session store is SQLite and must not be shared across concurrent tasks. The availability ledger, by contrast, is shared across the tasks of a configuration. This is the intended semantics: concurrency within a configuration competes for the same provider quota, exactly as a user's parallel sessions would.

Patch extraction and scoring. Model patches are extracted with `git add -A` followed by `git diff --cached`, excluding agent scratch files. Each patch is scored by the official `swebench.harness.run_evaluation` entry point immediately after the task completes, while the instance image is still resident, after which the image is deleted to bound disk usage.

Fault isolation. A failure in one task, whether an image pull error, a container fault, or an evaluator error, is recorded against that task and does not terminate the run. This matters at this scale: a single unhandled exception would otherwise discard the remainder of a configuration's daily quota, which is not recoverable until the next reset.

J Replication

Because no code or run records are released, this appendix states what a replication requires and what it should expect to match.

What must be re-implemented. An agent loop that issues one provider call per step and rebuilds each request from the full history, so that a step can be retried on a different model; the availability ledger and the scoring function of Section 3; the failure-class cooldowns of Table 1; and the failover discipline, including the distinction between a failure before any output has streamed and one after streaming but before a tool call has executed. The ablation switches of Appendix H name the module each mechanism occupies.

What must be held fixed. The protocol of Section 4: a fresh ledger per configuration, a start aligned to the daily quota reset, one binary across configurations, a fixed task shuffle so that a configuration limited to k tasks runs the first k of that order, per-task evaluation with the official SWE-bench harness, and the concurrency and time limit of Appendix I.

What should be compared. Absolute resolution rates depend on the provider set, the caps in force and the catalogue at the time of the run, none of which are stable. The quantities that should replicate are relational: that pooling and a single connected model differ in failure composition far more than in resolution, that the abandonment and time-budget differences are large while the resolution difference is not, that uniform backoff outperforms failure-class cooldowns at several times the request volume, and that a substantial fraction of failovers arise from catalogue drift rather than rate limiting. A replication that reproduces these relations under a different provider set would strengthen the claims; one that does not would localise them to our configuration, which Section K already identifies as a limitation.

K LIMITATIONS

Single seed and dependence between tasks. Every measurement derives from a single task ordering, and a second seed of the pooled and control configurations is queued as the most important addition to this work. Tasks within a configuration also share one quota ledger, so they are not independent draws and our intervals understate uncertainty by an amount we cannot quantify. Both caveats argue against interpreting the null resolution result as evidence of equivalence.

Choice of control and scope of the snapshot. Both criteria defining the control, our quality ranking and the pooled configuration's revealed usage, are internal to our system; we have no independent benchmark establishing that this is the strongest freely available model and do not claim it is. The control also began 3.5 h before a quota reset following heavy prior use, so its 41.9% is a lower bound. We contribute no learned policy, four of six mechanism ablations await fresh-pool re-runs, and the two valid ones are cold-start comparisons meriting replication. Free-tier limits change without notice: this is a single three-day window with eleven keys, on a pool where one provider served 71.4% of steps, and 248 of 300 instance images execute under x86-64 emulation, so wall-clock figures are machine-specific.