LazyCode: Compiling Interactive Intent into Batch Execution

A Query-Optimizer Architecture, Its Economics, and the Limits of Overnight Coding Agents

Gurunath Lunkupali Venugopal*

August 2026

Abstract

Coding agents call models the way a person would: one synchronous request at a time, at the full realtime price. Yet much of what agents are best at — burning down a backlog, mechanical migrations, test-coverage pushes — has nobody sitting there waiting for it, and every major provider already sells a cheaper product for exactly that situation: a batch API with a 24-hour completion window at half the price. LazyCode is a compiler that moves deferrable coding work onto that tier. A realtime model plans the work once; the plan then runs unattended as waves of self-sufficient batch calls, and memoization plus a three-window crash-recovery protocol make it safe to keep paying a provider while nobody watches. The design borrows deliberately from database query processing: an operator algebra, rewrite rules, physical execution waves, a write-ahead log. We are explicit throughout about what is implemented (20,369 lines, 440 tests) and what is still design. The cost argument is stated so a reader can check it on a napkin. Cache discounts only ever apply to *re-reading* input; a generation that has to happen bills at the full output rate on either lane, so batch's half price on output is unconditional. Input tokens are where the real contest is: an interactive agent re-reads its context every turn but pays a cached rate for the re-reads, roughly a tenth of list, while a batch call pays half of list and reads the context once per round — so batch wins on input only when the plan compresses T interactive turns into fewer than $T/5$ batch rounds. Wide waves relax that bar wherever batch and cache discounts stack. We then measure the whole argument with provider receipts on a public batch lane. Across HumanEval and SWE-bench tasks, the compiled plan cost 46× less than the same model driven by an interactive agent; the batch lane halved that again exactly as priced (receipted ratios 0.49–0.51 across three model families); and on the harder tier LazyCode's compiled calls resolved more issues than the interactive agent (8/10 against 6/10, identical across both price lanes). One upstream gap keeps general agent frameworks from following: they can already pause a run at a *tool* call and resume it later from saved state when the result arrives, but no framework can pause a run at a *model* call — so there is no seam at which to pool model requests into a provider batch. That seam — a deferred model request — is the framework maintainers' to add, not the providers': the batch APIs already exist and the durable backends already suffice. It is a small protocol change, and until a framework ships it, this pattern stays out of their users' reach.

1 Introduction

Every mainstream coding agent — Claude Code, Codex, and their descendants — shares one interaction contract: a loop of synchronous model calls, each awaited by a harness, each priced at the realtime rate. The contract is right when a person is watching. It is an unexamined default when nobody is: the overnight backlog task, the 400-file mechanical migration, the test-coverage push that nobody wants to babysit. For that work, latency tolerance is measured in hours, and the providers already sell a product priced for it —

*Code: `github.com/rajagurunath/lazycode`. Companion serving-side paper: Tidal (`github.com/rajagurunath/tidal`).

OpenAI Batches and Anthropic Message Batches both offer a 24-hour completion window at 50% of the online price.

The catch is structural, not financial. A batch call cannot ask a follow-up question, cannot read the result of the previous call in the same submission, and may take hours to return. An agent loop is the opposite shape: deeply sequential, incrementally contextualized, interactively repaired. Moving agentic work to the batch tier therefore requires *compiling* it: transforming a deep interactive loop into a shallow, wide, self-sufficient plan before the first batch call is made.

LazyCode is that compiler, and this paper is its story — told with database-systems vocabulary because the mapping is the design, not decoration. A realtime model emits a *logical plan*: a typed DAG over an eight-operator algebra of engineering work. An *optimizer* rewrites it. A *physical planner* lowers it onto provider batch APIs as barrier-synchronized waves, grouped like batched row processing. An event-sourced log makes the whole execution crash-safe and exactly-once in effect, the way a write-ahead log makes a database recoverable. Data systems spent forty years moving from batch to streaming; agent systems, born streaming, have never asked what moving the other way would buy.

What it buys is governed by one inequality (Section 3): the honest baseline is not an uncached agent but a prompt-cached one paying $0.1\times$ for repeated context, so batch's flat $0.5\times$ wins on input tokens only when the plan collapses T interactive turns into $R < T/5$ batch rounds. Output tokens carry no cache rate on either side — caching discounts re-reads, never generation — so batch's ~50% on them is an unconditional win. This single condition unifies the latency mechanism and the cost mechanism: the shallow-wide DAG is simultaneously what makes 24-hour latency tolerable *and* what makes the economics beat a well-engineered interactive baseline. We further show (Section 3.1) that where batch and prompt-cache discounts stack (provider-dependent; Section 3.1), the condition is width-dependent: a wave of $N \geq 3$ prefix-sharing calls needs only $R \cdot (0.05 + 0.95/N) < 0.1\,T$, which for wide waves relaxes toward $R < 2T$ — moving the design burden from *manufacturing depth-collapse* to *supplying width*, with consequences for generalization that Section 9 develops.

We make four contributions:

1. **An architecture** (Section 4): the query-processing transplant in full — operator algebra, rewrite rules with database analogues, content-addressed waves, and a three-window crash-safety protocol for paid third-party submissions. An implementation-truth account (Section 8) separates shipped machinery from declared vocabulary.

2. **An economics analysis, derived and then measured** (Section 3, Section 7): the $R < T/5$ derivation, its width-dependent refinement under discount stacking, the confrontation with `service_tier: flex` — the zero-architecture route to the same discount that any batch-tier argument must now name as its baseline — and a receipted three-arm study over a public batch lane that separates what the architecture saves from what the price list gives away.

3. **The self-sufficiency discipline** (Section 5): front-loaded context harvest under byte budgets, contract-constrained outputs, and the *assumption ledger*, which converts a blocking clarification into a decision flagged for morning review.

4. **A generalization verdict** (Section 9): a four-part criterion for when the plan-online, execute-on-batch pattern pays, a transfer analysis into Pydantic AI + DBOS and NVIDIA's NeMo Agent Toolkit, and deferred model requests identified as the missing upstream seam. Recommendation: build the wave executor as a library; do not port the planner.

Throughout, we distinguish three epistemic layers explicitly: what is *implemented and tested* (20,369 lines, 440 default-suite tests), what is *designed with plumbing in place*, and what is *vocabulary only*. Research prototypes usually blur these; a paper about honest economics should not. And where an earlier draft could

only derive its economics, this version also measures them: Section 7 reports a three-arm study on a public batch lane in which every dollar figure is a provider receipt.

2 Background: the two products and the honest baseline

Batch APIs. Both major providers accept a file (OpenAI) or list (Anthropic) of independent requests, promise completion within 24 hours, and charge half the synchronous price. Results arrive as a set and are kept for about a month. Three facts about this product shape everything downstream. No item can read another item's output, so anything submitted together must already be independent. Nothing can be clarified mid-flight, so every request must carry everything the model will need. And the scheduling is opaque: providers publish no latency percentiles, most items finish within an hour, but the tail is unbounded right up to expiry. A fourth fact bit us later: neither provider can cancel a single item, only a whole batch, and that one reshaped our hedging design (Section 6).

The honest interactive baseline. A naive comparison prices the interactive agent at list rates and reports 60–85% savings. That baseline is a strawman: production agents use prompt caching, under which repeated context is billed at $0.1\times$ base on cache reads. Our first design document made exactly this error; an adversarial review caught it, and the corrected analysis (Section 3) is one of this paper's contributions. The strawman survives in most published comparisons of batch pricing.

The newer complication. Since 2026, the choice is no longer binary. OpenAI's `service_tier: flex` charges batch rates on ordinary synchronous calls — slower, capacity-contingent (429 on shortage, unbilled), but requiring *zero* architectural change. And Anthropic's batch worker now runs the *server-side* agentic loop (web search, code execution, MCP connectors); provider materials have additionally advertised deeper per-turn iteration limits and larger maximum outputs on the batch path than the synchronous one — vendor-reported figures we have not independently verified, and which move with each release. The batch tier is thus no longer "the same API, slower and cheaper." It is architecturally different in both directions, and Section 9 weighs both.

3 The economics condition

The comparison to get right is a rent-versus-buy problem: the interactive agent re-reads its context every turn at the cheap cached rate, while the batch plan re-sends that context wholesale each round at half the full rate. Which side wins depends on nothing except how many turns compile into how many rounds. To make that exact, let a task require shared context C (repository slices, house rules, instructions). An interactive agent across T turns re-reads that context every turn; with prompt caching, the input bill is approximately $0.1\,C\,T$ (cache writes and the uncached first turn add small constants). This baseline is deliberately generous to the interactive side — it ignores cache-write premiums and the per-turn growth of the uncached conversation suffix, both of which raise the real interactive bill — so the condition below is conservative *against* batch. LazyCode front-loads the same context into each of R batch rounds at the batch rate: $0.5\,C\,R$, assuming cold caches across waves. Batch wins on input iff

$$0.5\,C\,R \;<\; 0.1\,C\,T \quad\Longleftrightarrow\quad \boxed{R < T/5}.$$

A 30-turn interactive task compiled to 2–3 batch rounds costs 1.0–$1.5\,C$ against the cached agent's $3.0\,C$; the same task compiled poorly, to 8 rounds, loses. Output tokens have no cached rate: caching discounts the re-reading of input, and a generation that must happen is billed in full on either lane, so batch's 50% discount

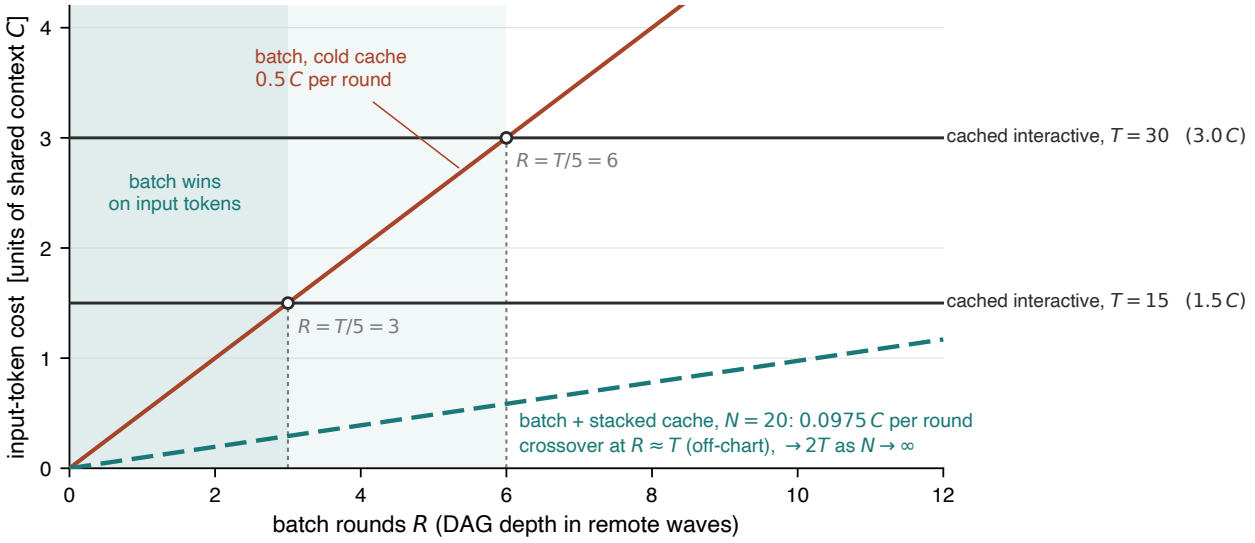


Figure 1: **The economics condition.** Input-token cost of completing one task, in units of its shared context C, as a function of batch rounds R. A prompt-cached interactive agent pays $\approx 0.1\,C$ per turn (horizontal lines for T=15 and T=30 turns). Uncached batch pays $0.5\,C$ per round: it beats the cached baseline only left of the R=$T/5$ crossovers. With batch and cache discounts stacked across a wave of N prefix-sharing items (dashed, N=20), the slope drops to $0.0975\,C$ and the crossover moves to $R \approx T$ (the $N \to \infty$ limit is $2T$): width substitutes for depth-collapse. Output tokens (not shown) favor batch unconditionally at ~50%.

on it is unconditional — and agentic coding is output-heavy. (Memoization, R10, is the one way outputs are "cached": an identical call is never re-bought at all. That saves whole calls; it does not discount a call that runs.) The blended saving is therefore "~50% on output plus a conditional input win": 30–70% against a well-cached baseline, and the widely-quoted 60–85% only against an uncached one. The design's own kill condition follows: if realized `rounds_per_node` degrades toward interactive depth, the input economics invert.

3.1 Width changes the condition

The derivation above assumes batch calls never hit caches. Where batch and prompt-cache discounts *stack* — both providers' documentation implies it; we have not verified the line items — that assumption is wrong in the favorable direction, and Anthropic's 1-hour cache TTL exists precisely for workloads whose calls are separated by batch-scale queuing delays. Within one wave, N items sharing a hoisted prefix C pay approximately $C \cdot 0.5 \cdot 2$ once (a 1-hour cache write at the batch rate, Anthropic's pricing) and $C \cdot 0.5 \cdot 0.1$ for the remaining $N-1$ reads — *when caching is worth requesting at all*. Because the cache write costs 2×, a rational renderer caches only when the wave is wide enough to amortize it, giving a piecewise per-item cost:

$$\text{per-item input} \;\approx\; C\,R \cdot \min\!\left(0.5,\; 0.05 + \tfrac{0.95}{N}\right) \quad\Longrightarrow\quad \text{batch wins} \iff R \cdot \min\!\left(0.5,\; 0.05 + \tfrac{0.95}{N}\right) < 0.1\,T.$$

At N=1 the min selects the uncached branch and the condition is the original $R < T/5$; caching pays for $N \geq 3$; at N=20, $R < 1.03\,T$; as $N \to \infty$, $R < 2T$.

The arithmetic is Anthropic-specific (explicit cache-control with a 2× 1-hour write); OpenAI's automatic positional caching has no write premium, making its optimistic form closer to $R(0.05 + 0.45/N)$ if the discounts stack there — a provider-by-provider question the benchmark must settle empirically. Two caveats

we owe the reader: our own adapter does not yet request the 1-hour TTL (it sends the default 5-minute `ephemeral`; the upgrade is a one-line change, listed in Section 11), and whether the 50% batch discount applies to the cache-read and cache-write *line items* is implied by provider "stacking" language but has not been verified by us.

Two things follow from this arithmetic. The first is that *width is the real currency*: a coding agent has to manufacture width by splitting one task along its independence boundaries, but fleet workloads — evaluation sweeps, corpus processing — have width by construction, which is the key fact behind Section 9's verdict. The second is a caution. Cache hit rates are best-effort (providers cite anywhere from 30% to 98%), and there is a concrete mechanism for missing the ideal: a cache entry only becomes readable after the request that writes it has begun returning, so wave items the provider happens to schedule concurrently can *each* pay the write, degrading one-write-and-$(N-1)$-reads toward N writes. The honest statement is a sensitivity range, not a point estimate, and within-wave hit rate belongs in the benchmark suite as a first-class metric.

3.2 What ten dollars buys

The conditions above are stated in token multiples; a reader wants dollars. So fix a concrete workload and price it at published list rates.[1]

One unit of work W: an overnight test-coverage pass over 12 modules of one repository. Shared context $C = 8{,}000$ tokens (execution role, repository outline, house rules) is byte-identical across items; each module contributes $u = 4{,}000$ tokens of unique context and receives $o = 2{,}000$ tokens of generated diff. The interactive agent works the modules sequentially over $T = 36$ turns (three per module) with prompt caching at a *generous* 100% hit rate — C written once and read 35 times, each module's u written once and read twice. LazyCode compiles the same unit into one wave of $N = 12$ items ($R = 1$ remote round per node) plus a free local VERIFY, and, conservatively, assumes *cold* batch caches: none of Section 3.1's stacking upside is booked. Both sides emit the same 24,000 output tokens — same model, same artifacts, same verifier — so the comparison holds quality fixed and varies only the tier.

[1] Anthropic, *Pricing*: `https://platform.claude.com/docs/en/about-claude/pricing` — Claude Haiku 4.5 at $1.00/Mtok base input, $1.25 five-minute cache write, $0.10 cache read, $5.00 output; Message Batches at $0.50/$2.50. OpenAI, *Pricing*: `https://developers.openai.com/api/docs/pricing` — gpt-4o-mini at $0.15/Mtok input, $0.075 cached input, $0.60 output; Batch API at $0.075/$0.30. Both accessed 12 August 2026; both providers publish the batch tier as 50% off input *and* output. Prices move, so `bench/cost_report.py` carries the same sheet declaratively and the arithmetic can be re-run against a current one.

Table 1: **One workload unit W, priced at published list rates.** Priced arithmetic over the cost model, *not* a measured invoice — no provider was billed for this table. The interactive column is prompt-cached at a generous 100% hit rate; the batch column assumes cold caches. "Input tokens billed" counts every token that meets a meter, which is why the interactive side pushes 3× the input volume for the same work: it re-reads C on every turn.

	Claude Haiku 4.5		gpt-4o-mini	
Per unit W	interactive	batch	interactive	batch
Input tokens billed	432 k	144 k	432 k	144 k
Output tokens billed	24 k	24 k	24 k	24 k
Input cost	\$0.1076	\$0.0720	\$0.0366	\$0.0108
Output cost	\$0.1200	\$0.0600	\$0.0144	\$0.0072
Cost per unit	**\$0.2276**	**\$0.1320**	**\$0.0510**	**\$0.0180**
Units bought by \$10	43.9	75.8	196	556
Saving	**42.0%**		**64.7%**	
\$10 of interactive work costs	**\$5.80**		**\$3.53**	

So: \$10 spent online-only buys 43.9 units of W on Haiku 4.5; the same \$10 spent through the batch tier buys 75.8 — or, equivalently, the 43.9 units cost \$5.80 instead of \$10. The table also shows two things the input-side model alone cannot. Output dominates the bill: it is 53% of the interactive total and 45% of the batch total, and it sits exactly where the 50% discount is unconditional, so the contested input win rides on top of a saving that is mostly not contested at all. And the size of the win turns out to be a property of the *baseline's* cache discount rather than of batch itself: gpt-4o-mini's cached input is 50% of list, not the 0.1× Anthropic-style read the derivation in Section 3 assumes, so its prompt-cached baseline is weaker and batch looks better against it. Quoting only the larger number would quietly reproduce the strawman of Section 2.

Everything above is priced arithmetic over the cost model, not a measurement. The instrument that does meter it ships — `bench/cost_report.py` runs one workload through both tiers and prices the metered ledger (Section 8) — and a real-provider spend measurement is still the named next step.

3.3 The flex-tier confrontation

OpenAI's flex tier delivers batch pricing on synchronous calls: one string, no architecture, minutes of extra latency instead of hours. Any defense of the batch-compilation architecture must therefore rest on what flex does *not* provide: the 24-hour window that potentially buys schedulable width (waves timed for cache stacking, subject to the hit-rate caveats of Section 3.1), the batch path's vendor-reported loop-depth and output-size advantages, cross-run pooling, and — on the self-hosted side — the co-serving economics of the companion system (Tidal), where the batch tier monetizes idle capacity a flex-style tier would still consume at peak. For pure per-call discount on an interactive loop, flex dominates; this paper's architecture earns its complexity only where those additional properties are the point.

4 Architecture

4.1 The logical plan: an algebra of engineering work

Eight typed operators form the planning vocabulary: EXPLORE, DECOMPOSE, GENERATE, EDIT, VERIFY, JUDGE, REDUCE, GATE. Only GENERATE and EDIT carry a `context_spec` (what to harvest) and an `output_contract` (what shape must come back) — they are the operators that spend money and mutate the

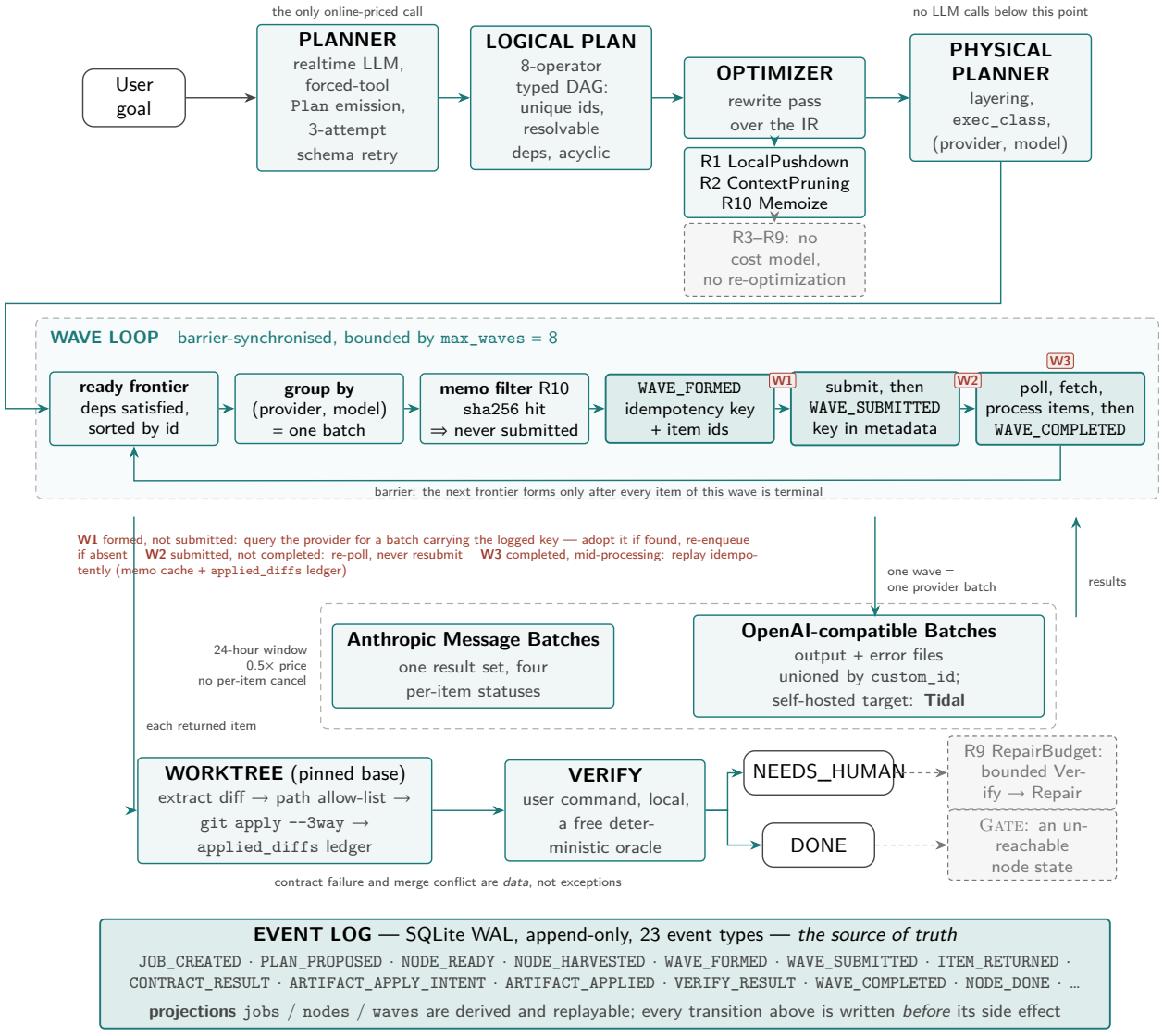


Figure 2: **The LazyCode pipeline.** Solid/accent components are shipped code; dimmed components are designed with plumbing but not yet live (Section 8). The event log is the source of truth; `jobs`/`nodes`/`waves` tables are replayable projections. W1–W3 mark the three crash-safety windows of Section 4.4.

tree. The plan is a Pydantic-validated DAG: unique ids, resolvable dependencies (including `"node.*"` fan-out patterns), acyclicity by iterative DFS. Critically, the planner cannot drift from this schema, because the planning prompt's tool definition *is* `Plan.model_json_schema()` and the algebra documentation shown to the model is generated from the model fields themselves; a validation error is fed back verbatim for up to three repair attempts.

The DAG-shape thesis is executable prompt text, quoted in full because it is the paper's mechanism in one sentence:

> *"Structure the work as a shallow, WIDE DAG (few dependency layers, much fan-out) because wall-clock is proportional to DAG depth: split independent work (per-file, per-module) into sibling nodes at the same layer instead of chaining it."*

4.2 The optimizer: rules with database analogues

Table 2 lists the ten rewrite rules with their database lineage. Two rewrites ship today, plus R10 — an execution-layer memoization rather than a plan rewrite; the table's Status column is load-bearing (Section 8).

Table 2: Optimizer rules, their database analogues, and implementation status. "Plumbed" = data model and call paths exist; the decision logic does not.

Rule	DB analogue	Status	Semantics
R1 LocalPushdown	predicate pushdown	**shipped***	planner-flagged EXPLORE executes locally, free, no LLM round (*the flag is trusted, not analyzed, and local execution today yields a scope artifact rather than tool output — Section 8)
R2 ContextPruning	projection pruning	**shipped**	nodes touching ≤2 literal files drop the repo map from their context
R3 FanoutWidening	join reordering	prompt-level	split coarse nodes along independence boundaries; today width comes from the planner prompt, not a rewrite
R4 PrefixFactoring	common subexpr. elim.	partial	shared context hoisted to a cache-hinted prefix block; per-call today, not per-wave
R5 ModelTiering	access-path selection	designed	cheapest model whose predicted verify-pass rate clears threshold; escalate on failure
R6 Vectorize	batched row processing	plumbed	pack k homogeneous tasks into one call; `call_items` maps 1 call to k nodes (k=1 today)
R7 Speculate	branch prediction	designed	submit enumerable continuations as siblings — restricted to *remote*-decider branches after review showed the local-decider case saves zero waves
R8 HedgeInsertion	hedged reads	designed	stall detection + realtime duplicate for critical-path items
R9 RepairBudget	—	designed	bounded Verify→Repair loops, then NEEDS_HUMAN
R10 Memoize	materialized views	**shipped**	`sha256(model, prompt, mode, sample_idx)` keys an exact-reuse cache; identical calls never re-bill

4.3 Wave formation and the barrier ruling

The scheduler runs a bounded loop over the *ready frontier*: nodes whose dependencies are all successful, sorted by id for deterministic apply order. Remote frontier nodes group by (provider, model); each group becomes exactly one provider batch — a *wave*. Waves are barriers: the whole frontier submits, the orchestrator polls to completion, results are processed, and only then does the next frontier form. Choosing barriers over rolling flushes was the design review's most contested ruling, and barriers won for two reasons that compound. With them the cost model stays literal — wall-clock is simply depth times wave latency, nothing hidden — and the acceptance test stays countable. Rolling execution is deferred, not rejected.

Wave identity is *content-addressed*: the idempotency key is a hash of the rendered items plus a flush ordinal recovered by scanning the durable log — not an in-memory counter — so a crashed-and-resumed orchestrator derives the same key and can adopt its own orphaned submission.

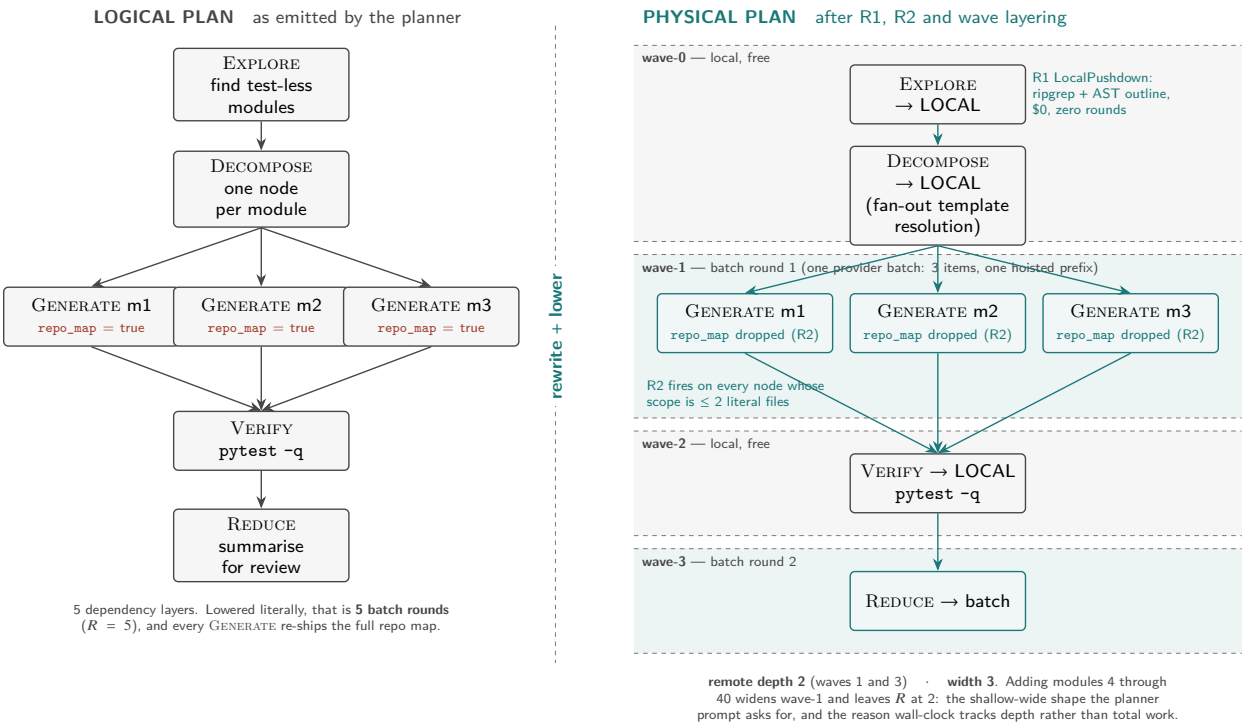


Figure 3: **A plan, compiled.** Left: the planner's logical DAG for "add tests for three modules." Right: after rewriting and lowering — the EXPLORE is pushed to free local execution (R1), the DECOMPOSE resolves locally as a fan-out template over EXPLORE's bindings (no LLM round; a judgment-requiring DECOMPOSE would add one), trivially-scoped GENERATE nodes shed their repo map (R2), and layering yields two remote rounds regardless of module count: depth stays constant while width scales.

4.4 Crash safety: three windows, three recoveries

A batch submission is a paid, remote, irrevocable side effect, and the system treats it with WAL discipline. `WAVE_FORMED` — carrying the idempotency key and item ids — is durably logged *before* any provider call; `WAVE_SUBMITTED` after; `WAVE_COMPLETED` only after every returned item is fully processed. Each window has a distinct recovery: (W1) formed-not-submitted resolves by querying the provider for a batch bearing the logged key — found means adopt, absent means re-enqueue; (W2) submitted-not-completed

re-polls and never resubmits; (W3) completed-mid-processing replays idempotently, because item results route through the memo cache and diff application checks a content-hash ledger before touching the worktree. Independent belts and braces: provider-side metadata, log-derived keys, the `applied_diffs` ledger, and a `UNIQUE(memo_key)` constraint on the spend cache. This is the strongest-implemented subsystem (Section 8), and deliberately so: in an architecture whose entire premise is unattended execution, recovery is the product.

5 From interactive intent to self-sufficient calls

Converting interactive intent into something that can run unattended means replacing three things the interactive loop provided for free: the ability to look something up mid-task, the ability to be corrected, and the ability to ask. One mechanism answers each.

Front-loaded harvest under byte budgets. Context is gathered *before* submission, deterministically: an AST-derived repository outline (demote-largest-then-drop under an 8 KB budget — breadth survives, detail degrades), the referenced files (60 KB each, 400 KB total, overflow explicitly listed as omitted), and house rules resolved from a fixed priority list of convention files. Determinism matters twice over: identical repo state yields byte-identical context, which is what makes the memo key (R10) an exact-reuse cache rather than a heuristic one.

Contracts on the way back. A GENERATE/EDIT node declares its output contract — today, a unified diff confined to a glob allow-list. Validation is defense-in-depth ordered: path normalization rejects absolute paths and traversal *before* glob matching (because `fnmatch`'s `*` happily matches `/` and `..`), then a three-way `git apply` against a pinned-base worktree, with rollback on conflict and a content-hash ledger making application idempotent. Contract failure and merge conflict both route to NEEDS_HUMAN — failure is data, not an exception.

The assumption ledger. The prompt contract states it plainly: *"You cannot ask questions — when you must make a judgment call, make the most reasonable choice and record it under an 'Assumptions:' heading."* The planner records its own scope-level judgment calls in the plan; executors append theirs after each artifact. What would be a blocking clarification in an interactive loop becomes a reviewable data artifact in the morning report. This is the interaction-model transplant in miniature — and, we will argue, the single most portable idea in the system (Section 9).

The tuning surface. The parameters that actually govern how intent decomposes today: the trivial-scope threshold for context pruning (2 files), the four harvest byte budgets, `max_waves` (the depth ceiling, default 8), the uniform (provider, model) assignment (which makes wave count equal DAG depth), sampling temperature 0 (which makes memoization sound), and the planner's schema-retry budget. A second tier — the cost slider λ, per-job budgets and deadlines, hedge timers, vectorize k — is parsed and stored but not yet consumed (Section 8); we list it as designed surface, not live control.

6 Provider reality

Building against the real batch APIs reshaped the design three times, and each lesson is worth writing down because none of them appears on a pricing page.

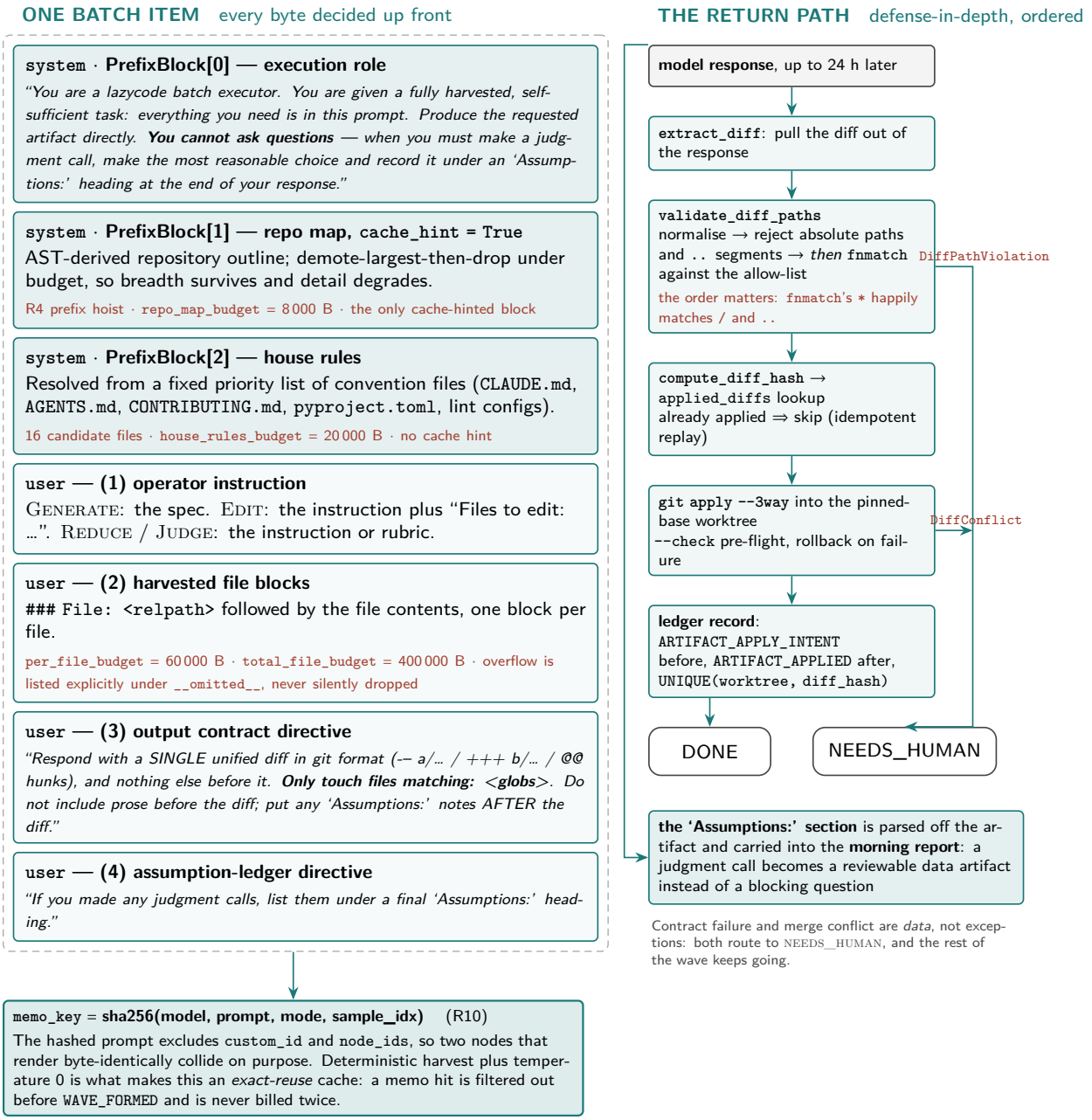


Figure 4: **Anatomy of a self-sufficient call.** Every byte the model will see is decided before submission, under explicit budgets; the output contract constrains the return shape; judgment calls come back as a structured *Assumptions* section feeding the morning report rather than blocking execution.

The first lesson is that per-item cancellation does not exist. On both providers you can cancel a whole batch or nothing. The original hedging design quietly assumed you could kill one stalled item; it had to be rebuilt as duplicate-and-discard, where a realtime copy races the stalled batch item, the first result to arrive wins at the node level, and the loser is billed and thrown away.

The second is that idempotency has to be visible to the provider, not just recorded locally. Both adapters therefore stamp the submission key into the batch's own metadata, so that a client which crashes between submitting and logging can later ask the provider whether its batch already exists. On the OpenAI wire this is a native field, and the round trip is verified end-to-end against a live gateway. On Anthropic it rides an `extra_body` escape hatch whose round-tripping is not documented, and our recovery coverage there is mock-level — so until it is verified live, the Anthropic W1 recovery should be read as "re-enqueue," with adopt-orphan guaranteed only on OpenAI-compatible providers.

The third is that the two wire protocols differ exactly where it hurts. Anthropic streams one result set with four per-item statuses. OpenAI returns two files, output and error, that must be unioned by `custom_id`; it has no per-item expired status, which must instead be inferred from batch state and count remainders; and it silently omits cancelled items altogether. The OpenAI-compatible adapter is also what lets LazyCode execute against *self-hosted* batch tiers — including Tidal, the companion system that co-serves batch work on the same GPUs as online traffic, closing the loop from client intent to self-owned capacity.

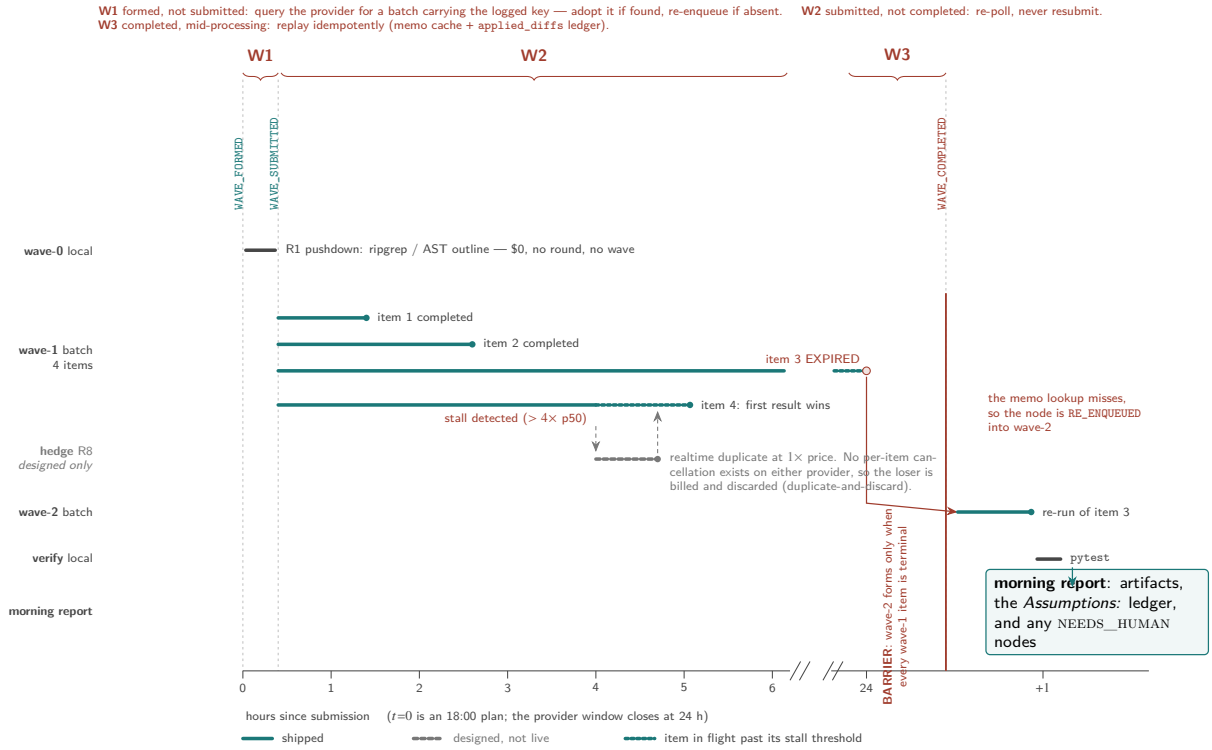


Figure 5: **Wave execution over a night.** Barriers separate waves; within a wave, items complete independently. The hedge path (dimmed: designed, not yet live) races a realtime duplicate against a stalled item under duplicate-and-discard semantics — the redesign forced by the absence of per-item cancellation. Expired items re-enter the next wave through the memo check. W1–W3 are the crash windows of Section 4.4.

7 Measured economics: receipts from a public batch lane

Between drafts of this paper, OpenRouter shipped a beta Batch API: one endpoint in front of the closed-model batch lanes, billing half the sync per-token price for the same model, with a 24-hour window and inline results. This gave the measurement campaign of Section 8 a metered public target months before the self-hosted one, and we took it. A fourth adapter (~300 lines plus 25 tests) speaks the wire; everything upstream — planner, optimizer, waves, store, verifier — is the code already described. All dollar figures below are *receipts*: the provider-reported `usage.cost` of each batch and the per-generation billing records of each sync call, reconciled against the account's own usage meter — not list-price arithmetic.

Setup. Every task runs three ways, and the reason there are three arms rather than two is the heart of the method. The first arm is the baseline this paper promised to be honest about: Claude Code itself, run headless, one session per task, routed through the same provider — a real agent loop, free to read files, run tools, and iterate. The second arm takes the single-shot calls that LazyCode compiles and executes them at the ordinary synchronous price. Nobody would ship this arm as a product; it exists so that the two possible sources of any saving can be told apart. Whatever it saves relative to the agent is the work of the compilation alone, with no discount involved; whatever the third arm saves relative to *it* is the work of the price lane alone, with no architecture involved. The third arm is LazyCode proper: the same calls, submitted as one wave to the batch lane. Prompts are byte-identical across all three arms and a single mechanical verifier judges them all, so quality comparisons are like-for-like. The models are three closed families whose batch lane is a clean 50% of sync (Claude Haiku 4.5, GPT-5.4-mini, Gemini 3.7 Flash). The fan-out tier is HumanEval, a fixed 50-problem subset verified by its own assertion suites; the agent may iterate as it likes, the single-shot arms get exactly one attempt. One seed, temperature 0. Raw responses, batch ids, generation ids and ledgers are archived in the repository.

Table 3: Fan-out tier (HumanEval subset-50), receipted. "sync" and "batch" are the *same* compiled single-shot requests on the two price lanes (only "interactive" is a different execution model — the agent loop); "x/50" is the verifier pass count; $ is the provider-charged total for all 50 problems. The batch:sync cost ratio is what the invoice says, not what the price page promises.

model	arm	pass	receipted $	batch:sync
Haiku 4.5	interactive	49/50	3.3113	—
Haiku 4.5	sync	46/50	0.0721	—
Haiku 4.5	batch	47/50	0.0367	0.509
Gemini 3.7 Flash	sync	47/50	0.0552	—
Gemini 3.7 Flash	batch	49/50	0.0281	0.510
GPT-5.4-mini	sync	46/50	0.0520	—
GPT-5.4-mini	batch	47/50	0.0255	0.491

What the receipts say. The most important number in Table 3 is not the discount. The agent was charged $0.066 per solved problem; the same model, handed a self-sufficient single-shot prompt, solved problems for $0.0014. That 46× gap has nothing to do with batch pricing. It is the cost of the agent's way of working — tool traffic, file round-trips, the growing conversation it carries from turn to turn — and it is precisely what the machinery of Section 5 compiles away. The batch lane then does what the price page says it does, which is itself worth reporting, because until this study the paper could only cite the price page: identical request bodies cost 50.9% of sync on Haiku, 51.0% on Gemini, and 49.1% on GPT-5.4-mini, taking the blended agent-to-LazyCode ratio to 90× on this tier. Quality moved the way the same-weights argument predicts it

should. The batch arm matched or beat the sync arm in every family, and the agent's ability to iterate bought it two extra solves over the one-shot arms (49 against 46–47) — a real advantage, honestly priced at about $0.065 per extra solve, and the gap the repair roadmap (Section 4.4) exists to close.

Running the study also surfaced two things documentation could not have told us. Batch turnaround turns out to be a property of the provider, not of the lane. Every Anthropic- and Google-routed batch, from 2 items to 50, finished within nine minutes of submission; the identically shaped OpenAI-routed batch sat at zero completions for its first 2.4 hours and finalized at 3.5 — a 25× spread on one workload. The 24-hour window is the contract, but the realized tail is whoever you were routed to, which is exactly why LazyCode prices the deadline rather than the typical case (Section 3). The second thing: an agent's own cost telemetry is not a receipt. The CLI reported $15.84 of list-price usage for sessions the provider actually billed at $3.31. Cache discounts and below-list routing account for the 4.8× gap, and they are the reason every number in Table 3 comes from the biller rather than from the client.

The pipeline run, end to end. The benchmark tiers exercise the economics, but the claim this system actually makes is different: that it can take an interactive-shaped goal and carry it through to applied, verified code over a batch lane with nobody watching. To exercise that claim we ran the `add-type-hints` fixture task through the shipped pipeline against the live lane. The planner's realtime call proposed a plan; one wave formed and was submitted; three items came back; all three passed their verify contracts; three diffs were applied; the repository-level verify ran green. JOB_DONE arrived 193 seconds after the goal, on a wave receipt of $0.00309, and the store's append-only event log — every state transition from PLAN_PROPOSED to JOB_DONE — is published with the run as the evidence artifact.

Getting there took two attempts, and the failure taught us more than the success. The first attempt crashed mid-poll, because the provider's just-created batch was not yet visible to reads and the adapter treated the resulting 404 as fatal. Recovery then behaved exactly as Section 4.4 claims it should: on resume, the W1 path found the already-paid batch by its idempotency key, adopted it, and finished the job with no duplicate spend — the receipts show a single batch. The lesson, that a freshly created batch can lag its own readability, is now a bounded retry in the adapter. The same wave exposed a second gap. Every diff passed its verify contract and then failed to *apply*, because the model wrote unified diffs with wrong hunk line-counts, which `git apply` rejects before it even looks at the content. Teaching the apply path to fall back to `--recount` (keeping the exact path and its rollback semantics first) took the wave from zero of three diffs applied to three of three — the first accuracy improvement this system can attribute to a receipted production run rather than to a code review.

The deep tier. HumanEval flatters the compilation — the prompts are tiny and nothing depends on anything — so the study's second tier is the shape that should favor the agent instead: ten SWE-bench Verified instances from the benchmark's easiest difficulty band ("<15 min fix"), real repositories, resolution judged by the official docker harness. The single-shot arms get the *oracle* retrieval setting: the problem statement plus the one file the reference patch touches, from which the model emits the complete corrected file. (We compute the diff locally from that file, so models are graded on the fix rather than on their ability to write diff syntax.) The agent gets no such help and explores the checkout itself, which is both harder and fairer to its nature. The agent resolved 6 of 10 for $5.69, at 15.8 turns and $0.95 per resolved instance. The compiled call at sync price resolved 8 of 10 for $0.258. The same calls through the batch lane resolved the identical 8 of 10 for $0.129 — half of sync on the invoice, $0.016 per resolved instance, 59× below the agent. Table 4 puts the two tiers side by side. This result deserves a fair reading: oracle retrieval hands the one-shot arms the located file, and locating is part of what the agent's turns pay for, so on instances deeper than this band the iteration should start earning its cost back. But the direction is the finding. On well-scoped work, one compiled call with the right context in front of it beat fifteen turns of exploration — and the batch lane then halved the bill

Table 4: Online agent vs. LazyCode, head to head (Claude Haiku 4.5, receipted). "Quality" is the verifier: HumanEval pass count on the fan-out tier, official-harness resolved count on the deep tier. "Wall" is what one run costs in time: per-item mean for the sequential arms, whole-wave turnaround for batch (all items land together). The agent buys +2 solves on the fan-out tier and *loses* two on the deep tier; LazyCode pays 1–2% of the agent's bill on both. †Not a product mode: the ablation control — LazyCode's own compiled calls, byte-identical, executed at the full synchronous price so the compilation effect and the lane discount can be read separately.

tier	arm	quality	receipted $	$/solved	wall
fan-out	agent (Claude Code)	49/50	3.3113	0.0676	18 s/item
(HumanEval-50)	compiled, sync price†	46/50	0.0721	0.0016	2.2 s/item
	LazyCode (batch lane)	47/50	0.0367	0.0008	8.1 min/wave
deep	agent (Claude Code)	6/10	5.6923	0.9487	91 s/item
(SWE-bench ×10)	compiled, sync price†	8/10	0.2582	0.0323	39 s/item
	LazyCode (batch lane)	8/10	0.1291	0.0161	9.6 min/wave

without changing a single outcome.

What this still does not show. The study leaves the width economics of Section 3.1 unmeasured: the fan-out prompts were too small for cache stacking to register, and the deep tier ran one call per instance, so `rounds_per_node` stayed at 1 by construction on both tiers. Each cell ran on a single seed. SWE-bench beyond its easiest band, and the self-hosted lane, both remain open. Section 11 keeps the full list.

8 Implementation truth

The system is 20,369 lines of Python (9,265 in the package, 7,987 in tests, 3,117 in the benchmark harness) with 440 default tests (442 including 2 deselected e2e tests). Three findings from auditing our own code, reported because papers rarely do and reviewers always should.

The optimizer is an architecture, not yet an optimizer. The shipped rewrite layer is 200 lines implementing two rules; R2's entire rewrite is one boolean flip under a hardcoded two-file threshold, and R1 trusts a planner-supplied flag rather than analyzing answerability — moreover its "free local execution" currently emits a question-and-scope artifact, not ripgrep/AST output (the operator schema has no `context_spec` to harvest), so fan-out driven by a local EXPLORE cannot yet resolve. There is no cost model, no adaptive re-optimization at wave boundaries, and `Caps`-aware splitting is validated post-hoc by adapters rather than planned around. The query-engine *shape* — typed IR, rewrite pass, physical lowering, execution — is real and tested; the query-engine *intelligence* is the roadmap. Relatedly, wave count equals DAG depth today only because a single model makes (provider, model) grouping degenerate, and the statically-computed layer assignment is discarded in favor of dynamically formed, content-addressed waves.

The state machine is ~40% aspirational. Of 21 declared node states, 8 are unreachable (hedging, speculation, repair, gating, cancellation); of 23 event types, 2 are never emitted. We keep the full vocabulary in the schema deliberately — events are append-only and renaming is a migration — but the paper's figures dim what the runtime cannot reach.

Crash safety is the strongest subsystem. Every claim in Section 4.4 traces to tested code paths — on the OpenAI-compatible path; the Anthropic W1 adopt-orphan recovery is mock-level (Section 6): the three windows each have a dedicated recovery with a dedicated test, including kill-9-mid-wave resume that adopts the orphaned provider batch rather than double-paying. Where the rest of the system is honest scaffolding, this part is load-bearing today — which is the right inversion for an unattended-execution product.

Evaluation status. A benchmark harness ships — fixture tasks, the standard-benchmark tiers of Section 7, a pricing model, and a Claude-Code-CLI baseline runner — and Section 7 now reports a receipted real-provider study over a public batch lane. What remains derived rather than measured has narrowed to three items: the width economics' cache-stacking term, realized `rounds_per_node` on dependent shapes, and the self-hosted lane. The last of these has a meterable target waiting: Tidal, the companion system, recovers 69.3% of a node's steady-state offline ceiling at 1.18×/1.23× online p50/p99 in its own evaluation [11], and the natural next run is these same tasks end-to-end against it, with per-item token ledgers and within-wave cache-hit rates recorded along the way.

The measurement instrument predates the live study and still earns its keep as the controlled half of it. `bench/cost_report.py` runs one workload through both tiers from a single plan — same orchestrator, renderer and harvester, only the adapter differs — and meters a per-call token ledger split into shared prefix, per-item unique context, and output, priced from a declarative sheet of the list prices in Section 3.2. On metered mock workloads it reproduces the predicted per-item input factor on both branches of the $\min(\cdot)$: the cold-cache default observes the flat 0.5, and `--stack-cache` observes 0.3667 against $\min(0.5,\ 0.05{+}0.95/3)$ at $N{=}3$. Its quality-equivalence report is exact by construction, because both tiers replay the same fixture responses at temperature 0 — a plumbing check, not evidence about models. The evidence about models is Section 7.

9 Should this generalize? Frameworks, and an honest verdict

LazyCode is a coding CLI. The obvious question — asked of us, and by us — is whether plan-online/execute-on-batch belongs in general agentic frameworks: Pydantic AI with its durable-execution backends, NVIDIA's NeMo Agent Toolkit, LangGraph and kin. The answer we defend: *partially, and not as an architecture*.

The substrate exists; the seam does not. Durable-execution integration is a solved problem: Pydantic AI ships co-maintained DBOS, Temporal, Prefect and Restate backends behind a public capability interface, with cost-budget gating built in; DBOS alone provides exactly-once steps, checkpointed multi-day sleeps, deduplication ids (LazyCode's idempotency key, ready-made) and rate-limited queues — and its old Postgres objection has expired (SQLite is now the default).

What no framework has is a *first-class batch-API abstraction for model requests*: durable backends do wrap model calls in activities/steps (and Temporal can suspend turns on durable timers), but an activity-wrapped model call still completes as one synchronous unit — there is no protocol for terminating a run at a model call, pooling it into a provider batch, and resuming hours later from persisted history, the way Pydantic AI's deferred-*tool* protocol already does for tools. The single upstream change that would open this pattern is a `DeferredModelRequests` analogue of the existing deferred-tools protocol — terminate the run at a model call, resume from persisted history plus results. To be clear about who owns this: it is not a provider change (the batch APIs exist) and not a durable-backend change (their machinery suffices); it belongs to the agent-framework layer that owns the run loop. Pydantic AI is the natural first implementer — the deferred-tools protocol this would mirror is theirs, the transport machinery already ships, the extension is small, and an open issue in their tracker has waited since May 2025 for exactly this design.

The negative control. NeMo Agent Toolkit — the most explicitly engineered framework in the field — has a genuinely strong profiler (its common-prefix detector independently rediscovers R4 from the serving side) yet no durable *mid-run* suspension, no cost-tier routing, and no provider-batch integration. The gap LazyCode fills is real across the ecosystem, not an artifact of coding CLIs. The only prior attempts at agent-side batch use are two 2026 fleet-pooling prototypes — both dispatcher-level, both stopping short of barriers, memoization, deadline machinery, and crash safety. Evaluation harnesses that *do* support batch APIs explicitly warn against using them for agentic tasks. We found no prior system that combines barriers, memoization, deadline machinery, and crash safety behind the batch tier.

The criterion. The architecture pays exactly where a workload satisfies four conditions: *(a) the environment is snapshottable* — LazyCode pins a git commit and applies diffs into an immutable fork, so an 18:00 plan is still valid at 03:00; a CRM-writing or browser agent acts on a live world that moves during the wave, and deferred action against an unversioned environment silently invalidates its own premises; *(b) a cheap deterministic oracle exists* — pytest and compilers make VERIFY free and local; where the only verifier is an LLM judge, failure is discovered a wave late and the repair round erases the discount; *(c) width dominates depth* — structurally within the task, or by inter-run concurrency (Section 3.1); *(d) laxity is positive* — nobody is waiting. Coding is the canonical member of this class — versioned by git, oracled by test suites, fan-out-able by file — but it is a member, not the definition. Repo-scale migrations, IaC changes (`terraform plan` as oracle), corpus extraction under schema contracts, and evaluation/synthetic-data fleets all qualify. Interactive copilots, SLA-bound triage, and live-environment actuation never will; naming that boundary is a contribution, because a naive generalization into those classes would do real damage.

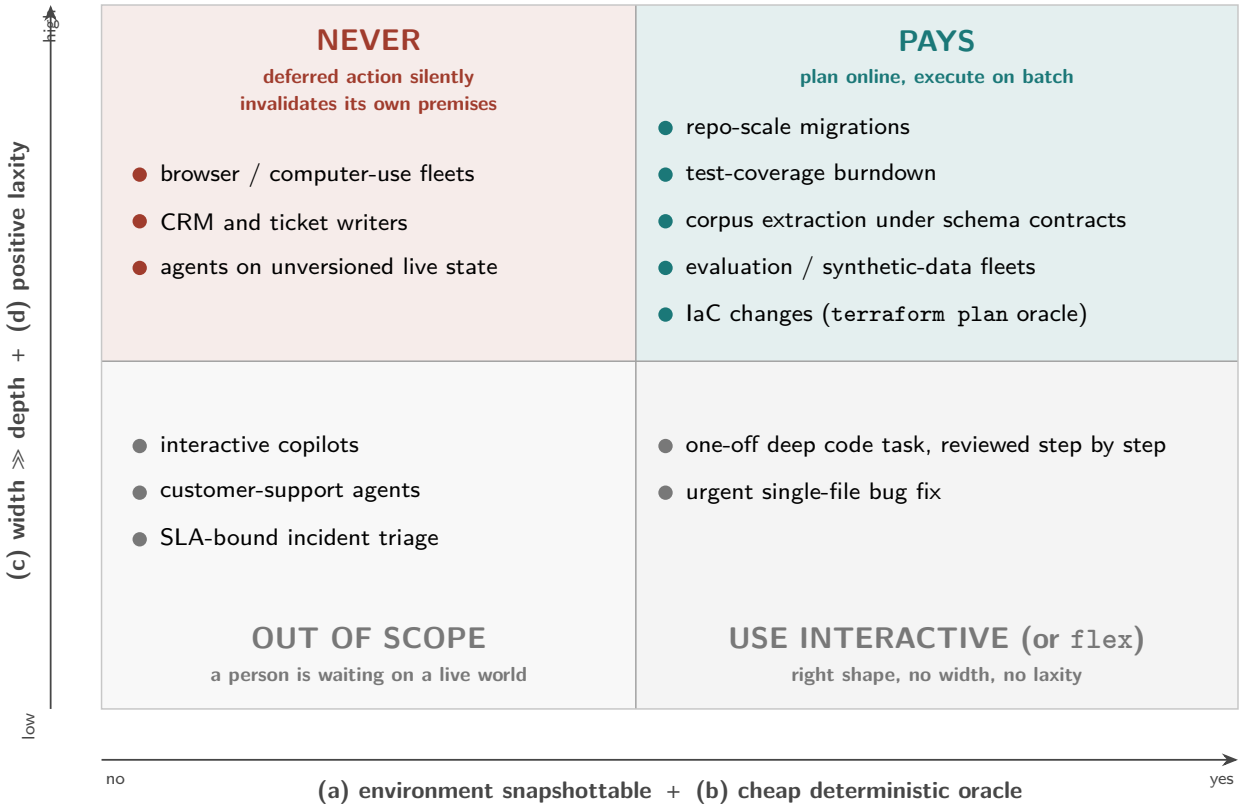
Transfer analysis and verdict. Going component by component, the split is clean. The *wave executor* transfers: barrier scheduling reinterpreted over N concurrent runs, memoization (R10 verbatim), prefix factoring (R4, worth more under discount stacking), the vectorization plumbing (R6), laxity-driven routing across {sync, flex, batch} with duplicate-and-discard hedging (R8), and the assumption ledger. Together they are worth building as a framework-agnostic library of one or two thousand lines, with thin DBOS and Temporal recipes and a Pydantic AI capability for cross-run pooling.

The *planner, the operator algebra, and the code-analysis rules* (R1–R3, R9, the harvester) do not transfer, and the reason is their content rather than their form: they are made of ripgrep, per-file independence, and free test oracles — irreducibly coding. Our recommendation, in ascending order of leverage: build the library; ship the pooling capability; and above all land the deferred-model-request protocol upstream. It is small, it is the true blocker, and it would open the pattern to every durable-execution framework at once.

10 Related work

Semantic operators and LLM query optimization. LOTUS, Palimpzest, Abacus, DocETL and Aryn establish cost-based optimization over declarative LLM pipelines; they own the operator-DAG-with-optimizer idea for *data* processing. LazyCode's distinction is the workload class — stateful, side-effecting, tool-using engineering work against a versioned environment with contract verification — and the batch-API execution substrate. FrugalGPT and RouteLLM give the cost-tiering lineage for R5 (the verify-pass-rate-gated variant is ours but unimplemented); Halo and Helium consolidate multi-agent queries into optimized DAGs with KV-sharing on *local* serving — the same instincts, the opposite deployment; sleep-time compute is adjacent (idle-time precomputation, not deferred pricing).

Coding agents. SWE-agent's measured trajectory depths ($\approx$13–15 turns) calibrate what T the economics condition must beat; OpenHands, Aider, and Agentless calibrate the interactive baseline space.



(a) LazyCode pins a git commit and applies diffs into an immutable fork, so an 18:00 plan is still valid at 03:00.
(b) `pytest` and compilers make VERIFY free and local; where the only verifier is an LLM judge, failure surfaces a wave late and the repair round erases the discount.
(c) width comes structurally from the task or from inter-run concurrency (§3.1); once batch and cache discounts stack, width is the currency.
(d) nobody is waiting.

Figure 6: **Where plan-online/execute-on-batch pays.** The four-part criterion as a map. The architecture's home turf is the upper right; the upper left is the danger zone where deferred execution silently invalidates its own premises.

Batch tooling and fleet pooling. Curator, Ray Data LLM, and evaluation harnesses batch *independent* calls; none run an agentic loop through the batch tier, and the leading eval framework warns against trying. Two 2026 fleet-pooling prototypes mark independent convergence on inter-run batching, minus the systems machinery.

Database lineage. The transplant's ancestors are explicit — System R's cost-based optimization, Volcano/Cascades rewrite architecture, ARIES-style write-ahead recovery, and Saga-style compensation for long-lived transactions — and the paper's claim is the transplant, not the machinery.

The serving side, and the claim. The companion Tidal paper supplies the self-hosted batch tier — deadline-contracted co-serving on unmodified vLLM — that closes the loop this paper's adapter opens. The unoccupied conjunction LazyCode claims: a barrier-scheduled, memoized, crash-safe, deadline-aware batch executor driving a stateful, contract-verified agent against a snapshottable environment.

11 Limitations

The headline economics are now measured against a provider invoice (Section 7), but at benchmark scale, on one seed, over one public lane; the dollar figures of Section 3.2 remain priced arithmetic, and no self-hosted-lane run exists yet. Realized `rounds_per_node` is not instrumented — the current counter can only report 1 — and within-wave cache hit rates, on which Section 3.1 leans, are best-effort and unbooked. Provider latency tails are unpublished, and the study showed why that matters: identical workloads finished in nine minutes on one provider and three and a half hours on another, so the overnight guarantee is "most nights, with a priced fallback," not an SLA, and a hedge that fires re-buys realtime pricing for that item. The optimizer intelligence, hedging, repair, model tiering, vectorization $k>1$, and per-item cost accounting are designed but not live; the state machine carries their vocabulary ahead of their arrival. The flex tier undercuts the pure-discount motivation on one provider today. Two code-level debts remain from review (a third, the diff-apply fragility, was found and fixed by the live run of Section 7): the adapter requests the default 5-minute cache TTL rather than the 1-hour TTL the width economics assume, and Anthropic-side batch-metadata round-tripping, on which orphan adoption rests there, is unverified against the live API; local EXPLORE also still produces a scope stub rather than tool output. And the architecture's preconditions — git-snapshottable state, free oracles — bound its domain honestly: this is a compiler for a particular, valuable shape of work, not a universal agent runtime.

12 Conclusion

Data systems learned to go from batch to streaming; LazyCode runs the lesson in reverse for agents, and finds the pieces already lying around: a planning model that can emit typed DAGs, providers selling half-price deferred execution, database machinery for making it crash-safe, and discount stacking that rewards width more than heroic depth-collapse. The system's honest state is a strong skeleton with two shipped rewrite rules. Its strongest-tested subsystem is recovery, and that is no longer only a test suite's opinion: it has now recovered a real crashed job over a real batch lane without paying twice. Its economics, which an earlier draft could only derive, now have first receipts (Section 7), including the one this paper most needed — the same verifier passing the same work at a 46× lower compiled cost before any discount is applied. The honest future is the rest of that campaign, the companion self-hosted batch tier among it, and a small upstream protocol change that would let durable-execution agent frameworks defer a model call. When we began, the compiler metaphor set a standard the project had not yet met: not that batch execution is architecturally workable,

which the shipped machinery and its crash safety argue on their own, but that the compilation is profitable, case by case, with the receipts to show it. The first of those receipts are now in the paper.

Acknowledgments

The design benefited from adversarial multi-model review; several of this paper's central corrections (the cached-baseline economics, the vacant-speculation example, the width-dependent refinement) were surfaced by reviews that set out to break the claims rather than confirm them.

References

[1] L. Patel et al. LOTUS: Semantic Operators for Bulk LLM Processing. VLDB 2025. arXiv:2407.11418.

[2] C. Liu, M. Russo, M. Cafarella et al. Palimpzest: Optimizing AI-Powered Analytics. CIDR 2025. arXiv:2405.14696.

[3] Abacus: Cost-Based Optimization for Semantic Operator Systems. arXiv:2505.14661.

[4] S. Shankar et al. DocETL: Agentic Query Rewriting for Complex Document Processing. arXiv:2410.12189.

[5] L. Chen, M. Zaharia, J. Zou. FrugalGPT. arXiv:2305.05176.

[6] I. Ong et al. RouteLLM: Learning to Route LLMs. arXiv:2406.18665.

[7] Halo: Holistic Agent-DAG Optimization for LLM Serving. arXiv:2509.02121.

[8] Helium: DAG-Consolidated Serving for Agentic Workflows. arXiv:2603.16104.

[9] K. Lin et al. Sleep-time Compute. arXiv:2504.13171.

[10] J. Yang et al. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv:2405.15793.

[11] G. Lunkupali Venugopal. Tidal: Co-Serving Online and Batch LLM Traffic under Deadline Contracts. 2026. doi:10.5281/zenodo.21905461. github.com/rajagurunath/tidal.

[12] Pydantic AI: Durable Execution (DBOS, Temporal, Prefect, Restate). pydantic.dev/docs/ai, 2026.

[13] DBOS: Durable Workflows. docs.dbos.dev, 2026.

[14] NVIDIA NeMo Agent Toolkit (v1.8). github.com/NVIDIA/NeMo-Agent-Toolkit, 2026.

[15] UK AISI. Inspect AI: Batch Mode. inspect.aisi.org.uk/models-batch.html.

[16] Anthropic. Message Batches and Prompt Caching documentation. platform.claude.com/docs, 2026.

[17] OpenAI. Batch API, Flex Processing, and Webhooks documentation. developers.openai.com, 2026.

[18] E. Sandler. Batch API is terrible for one agent. It might be great for a fleet. Apr 2026. github.com/erans/batching-harness.

[19] llmfleet: latency-budget routing to batched Anthropic calls. github.com/MukundaKatta/llmfleet, 2026.

[20] Bespoke Labs. Curator. github.com/bespokelabsai/curator.

[21] G. Graefe. Volcano — An Extensible and Parallel Query Evaluation System. IEEE TKDE, 1994.

[22] P. G. Selinger et al. Access Path Selection in a Relational DBMS. SIGMOD 1979.

[23] G. Graefe. The Cascades Framework for Query Optimization. IEEE Data Eng. Bull., 1995.

[24] C. Mohan et al. ARIES: A Transaction Recovery Method. ACM TODS, 1992.

[25] H. Garcia-Molina, K. Salem. Sagas. SIGMOD 1987.

[26] X. Wang et al. OpenHands: An Open Platform for AI Software Developers. arXiv:2407.16741.

[27] C. Xia et al. Agentless: Demystifying LLM-based Software Engineering Agents. arXiv:2407.01489.

[28] Apache Spark: Adaptive Query Execution. spark.apache.org.