INTELLIGENCE WAREHOUSE · RESEARCH

Intelligence Warehouse reduces token consumption by 96%

Most enterprise AI is built by putting the whole program inside the prompt. IW works differently. The orchestration runs as ordinary Python, and the model is called only at the points that genuinely need language reasoning. We simulated 1,000 enterprise tasks to see what that does to the bill. Even after giving the prompt-based version every advantage we could (full prompt caching included), IW processes 96% fewer tokens and runs at roughly an eighth of the cost.

Authors Akhil Singh · Nidhi Prabhu · Aniruddha Tammewar **Date** July 2026

Method Seeded Monte Carlo, n=1,000 **Reproducible** seed 20260729

95.97%	87.54%	24.8×	7.9 → 2.2
FEWER TOKENS PROCESSED PER TASK	LOWER RUN-COST AFTER CACHING THE BASELINE	TOKEN THROUGHPUT MULTIPLE	LLM CALLS PER TASK BASELINE → IW

There are two numbers here because they measure two different things. **Tokens processed** is the raw work the model does. **Run-cost** is what lands on the invoice, and we calculate it after giving the baseline full prompt caching, which cuts the price of its repeated tokens by 90%. IW comes out ahead on both.

01 Why prompt-based agents get expensive

A prompt-based agent tends to look cheap in a demo and then surprise you in production. The reason is baked into how it works.

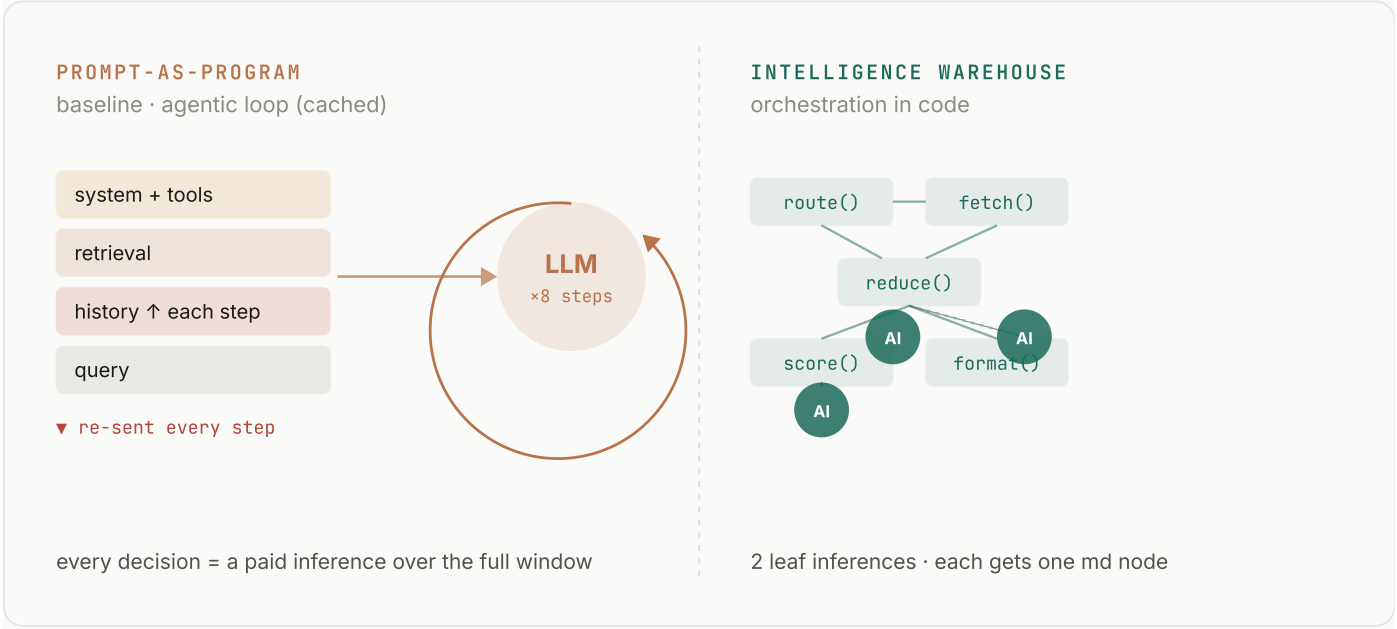
In a typical agent, the model runs the whole control loop *inside the context window*. Each step, it re-reads the system prompt, every tool schema, whatever the retriever returned, and the full conversation up to that point, then produces one more step and starts over. So a task that takes eight steps doesn't cost you eight prompts. It costs one prompt that keeps growing, charged again on every step.

Prompt caching helps here, but it doesn't fix the underlying problem. Caching makes the *re-read* tokens cheaper; it doesn't make the model stop processing them. Every cached token still runs through the model on every pass, you just pay about a tenth as much for it. At a few calls a day, none of this matters. At a million tasks a month, it's a real line on the bill.

IW starts from a different assumption: **the model is one component of the system, not the system itself.** The orchestration (routing, parsing, calculations, shuttling data between steps) is written as deterministic Python. Knowledge sits in a network of markdown files, and any given file is loaded only when an inference actually needs it. The model gets called at the leaves of that call graph, each time with just enough context to answer correctly. The question this paper sets out to answer is a simple one: **per completed task, how much does that actually save once you stop handicapping the comparison in the baseline's favour?**

02 Two ways to build the same thing

For the comparison to be fair, both systems have to be built well. The one on the left is a competent agent, not a strawman: it caches its static prefix, keeps retrieval tight, and loops until the task is done. On the right is the IW version.



Both systems call the model at the leaves, so that's not where they differ. The difference is everything *in between*. In the agent, the glue holding the steps together is tokens: every routing decision and every intermediate result passes through the context window and gets billed, again and again. In IW, that glue is just a function call stack. It runs on CPU, costs a fraction of a cent, and never touches the token meter.

03 Where the tokens actually go

The savings come from three separate things. It's worth pulling them apart, because that lets you check the math rather than take a single number on faith.

Mechanism 1: Control flow moves from tokens to code

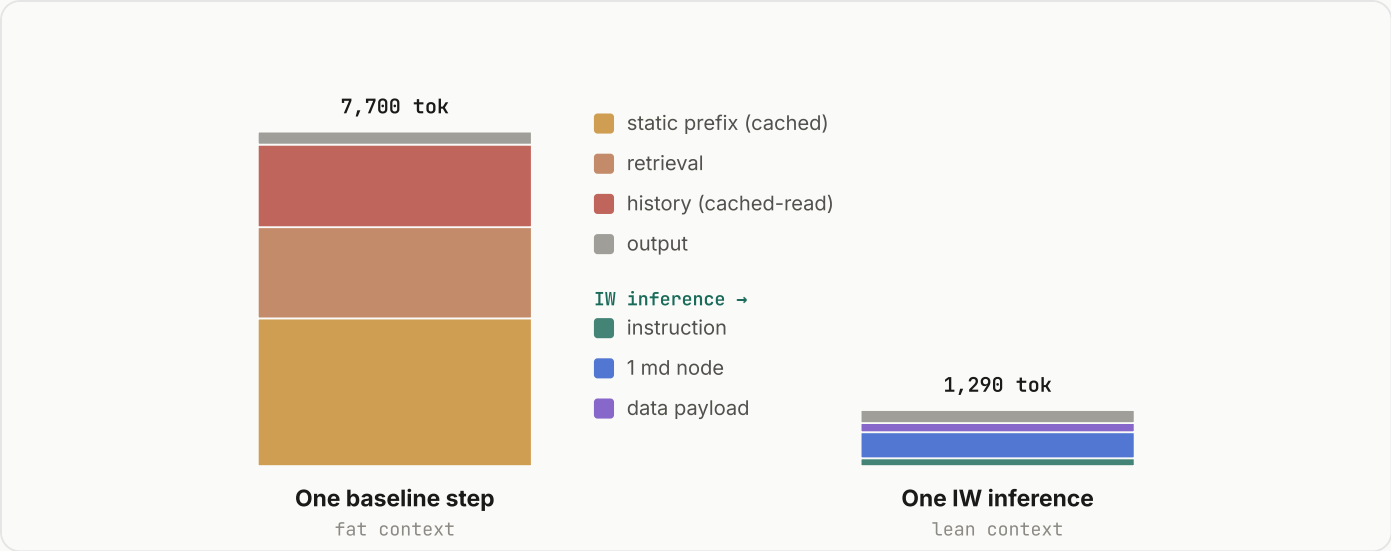
In an agent, the model *decides what to do next inside the context window*, so every decision is a paid inference over the whole working set. IW makes those decisions in code. Where the baseline takes ~8 model steps to work through a task, IW usually needs ~2 real inferences; the rest is deterministic code that never hits the meter. This is where most of the savings come from.

Mechanism 2: Context is loaded lazily, not defensively

A prompt system loads the window with everything it *might* need: the full system prompt, all the tool schemas, a generous helping of retrieved chunks, a few examples. IW gives a leaf inference one or two markdown files (the ones that inference actually needs) plus the data it has already fetched. You end up paying for the context you use rather than the context you might use.

Mechanism 3: History is never re-transmitted

The agent re-sends a growing transcript on every step, so cost climbs roughly with the square of the number of steps. IW keeps state in Python variables, so each inference stays small and flat. Even with the baseline caching that history at a tenth of the price, the re-reads pile up over a long task. IW never sends them in the first place.



Anatomy of a single call, drawn to scale from the model's mean draws. **Left:** one step of the cached agentic baseline: static prefix (cached), fresh retrieval, re-read history, output. **Right:** one IW leaf inference: a small instruction, one knowledge node, the data payload, output.

04 Method

Everything below comes out of one seeded simulation, `iw-token-economics-sim.py`, run over **1,000 tasks** drawn from five real IW verticals. Nothing in this paper is typed by hand. Each number is read straight from the script's output. The full model is reproducible, and we've published the per-task dataset alongside it.

How we handicapped ourselves

We set things up to work *against* our own conclusion in three ways:

- **The baseline gets full prompt caching.** Its static prefix is written once and read back at a tenth of the price after that, and its growing history is cached the same way. That's the most generous treatment the API allows.
- **IW gets no caching at all.** Every IW context is billed at full input price, start to finish. This makes IW look worse than it really is.
- **The headline is dollar-weighted.** We divide total IW cost by total baseline cost across all 1,000 tasks, so the expensive tasks carry the most weight and a pile of cheap, easy wins can't pad the average.

With all three of those in place, the number we report is closer to a floor than a ceiling.

The cost model

It's one cost function with three switches, and its two extreme settings are exactly the two architectures. Token flow is tracked in four buckets (fresh input, cache write, cache read, and output), priced at Claude Sonnet-class published rates.

```
# USD per 1M tokens input 3.00 output 15.00 cache-write 3.75 cache-read 0.30 cost =
fresh_in·3.00 + cache_write·3.75 + cache_read·0.30 + output·15.00 # divided by 1e6
```

The three switches map one-to-one onto the mechanisms in §03: `m1` control-flow→code, `m2` lazy context, `m3` no history. Setting all three off reproduces the cached agentic baseline; setting all three on reproduces IW. Because the endpoints share one function, the decomposition in §06 is exact.

The portfolio

Each task samples an archetype, then draws its parameters from documented triangular ranges: agent steps `S`, tools available `T`, retrieved chunks `K`, and the pivotal one, the fraction of steps that are genuine language inferences rather than deterministic work, which sets IW's leaf count `I`.

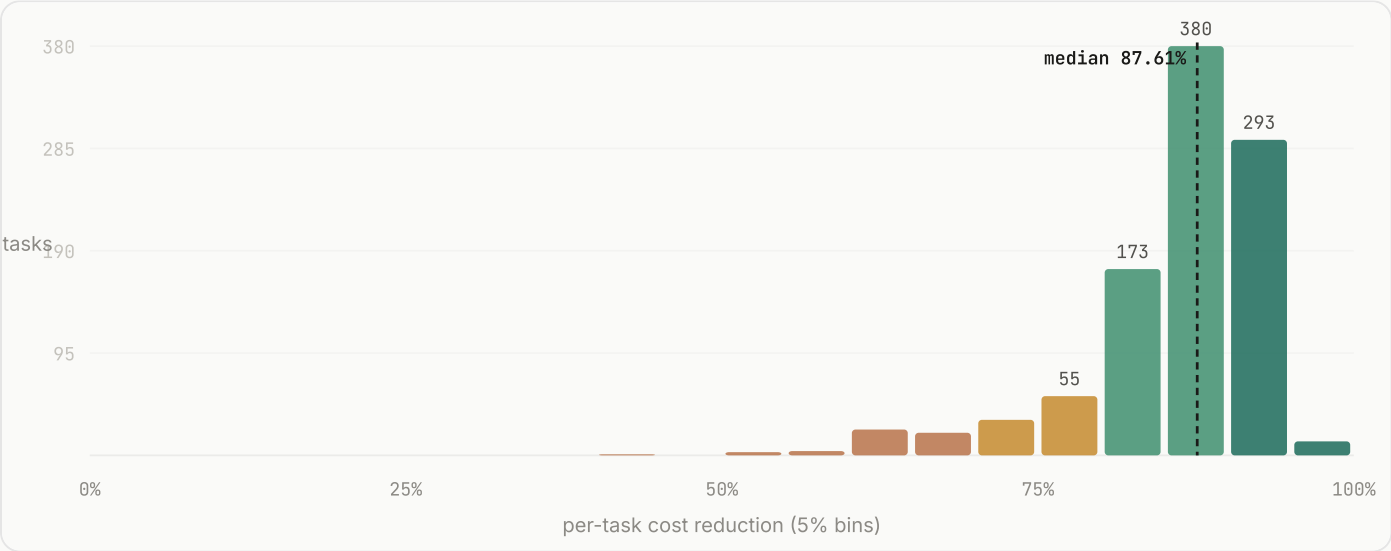
ARCHETYPE	SHARE	STEPS S	TOOLS T	CHUNKS K	INFERENCE:STEP
Retail replenishment	28%	4–12	8–20	3–8	0.15–0.40
Telecom rollout dispatch	22%	6–18	12–30	2–7	0.12–0.35
PE diligence query	15%	3–9	5–14	6–18	0.25–0.60
Supply-chain exception	20%	5–14	10–24	3–9	0.15–0.40
Field-ops query	15%	2–6	4–10	2–5	0.30–0.70

> **The assumption carrying the most weight** is the inference:step ratio, our claim that most agent steps are deterministic. We think it holds up (routing, tool selection, parsing and arithmetic genuinely don't need a language model), but it's still an assumption. §07 sweeps it across a fourfold range and shows what happens at every point.

05 Results

Across all 1,000 tasks, IW processes 95.97% fewer tokens and costs 87.54% less to run, and that cost figure is after the baseline has been given full caching.

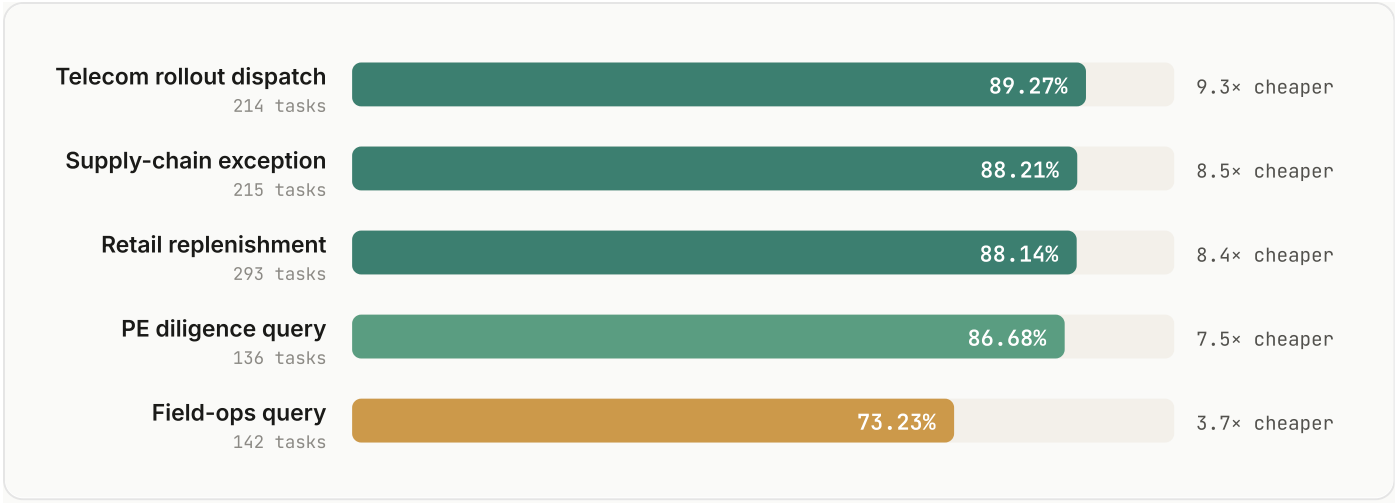
The two numbers differ for a straightforward reason. Caching makes the baseline's repeated tokens ten times cheaper, so its dollar cost drops faster than its token count does. That's why the token reduction (95.97%, an 24.8× multiple) is bigger than the cost reduction (87.54%, an 8× multiple). Both are accurate. We lead with the cost number because it's the one that shows up on the invoice, and the one we handicapped ourselves to calculate.



Distribution of per-task cost reduction across all 1,000 tasks. Median 87.61%, with the middle 80% of tasks falling between 76.5% and 92.56%. The left tail is real: simple field-ops queries don't have much control flow to remove, so they save the least.

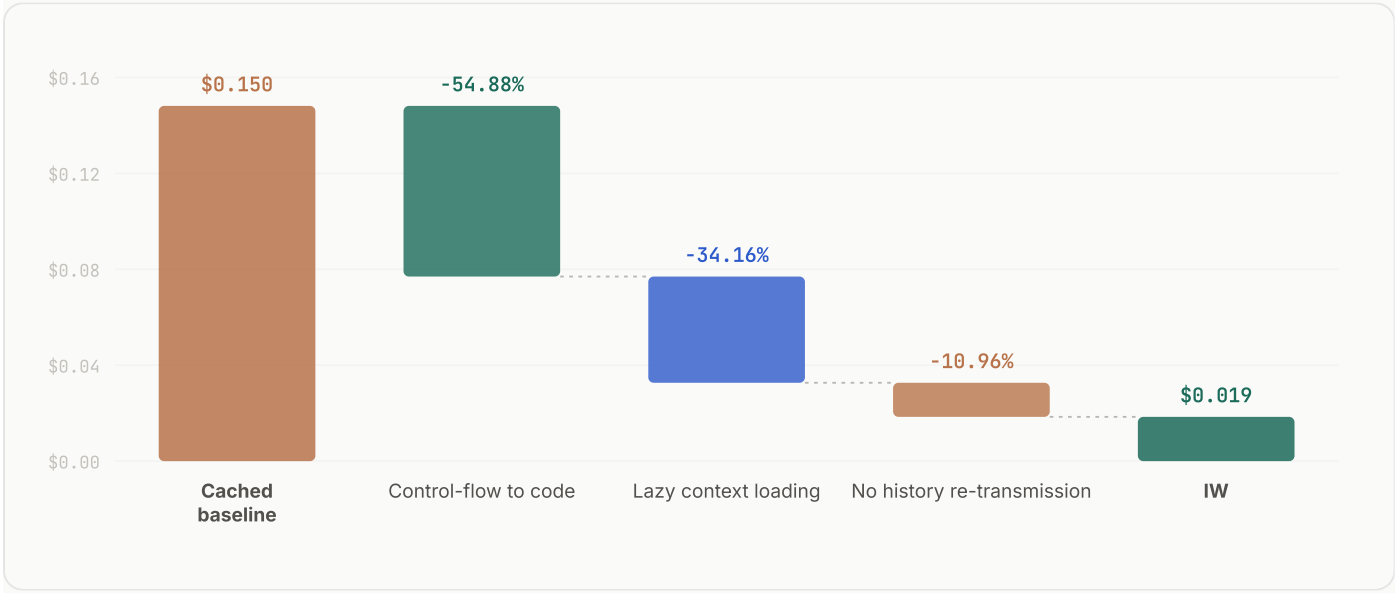
By vertical

How much a vertical saves depends on how much orchestration it involves. Telecom dispatch, with its long loops and large toolsets, saves the most. Simple field-ops queries save the least, which is what you'd expect.



06 Where the savings come from

A single headline percentage is easy to doubt and hard to check. Since the baseline and IW are just two settings of the same cost function, we can split the saving cleanly across the three mechanisms using their Shapley values: the average contribution each one makes across every possible order you could apply them in.



Mean cost per task dropping from the cached baseline to IW, one mechanism at a time (Shapley-attributed, so the order doesn't matter). **Control-flow→code** accounts for most of it. **Lazy context** is next. **No-history** is real but smallest, because the baseline's caching had already taken most of the sting out of it.

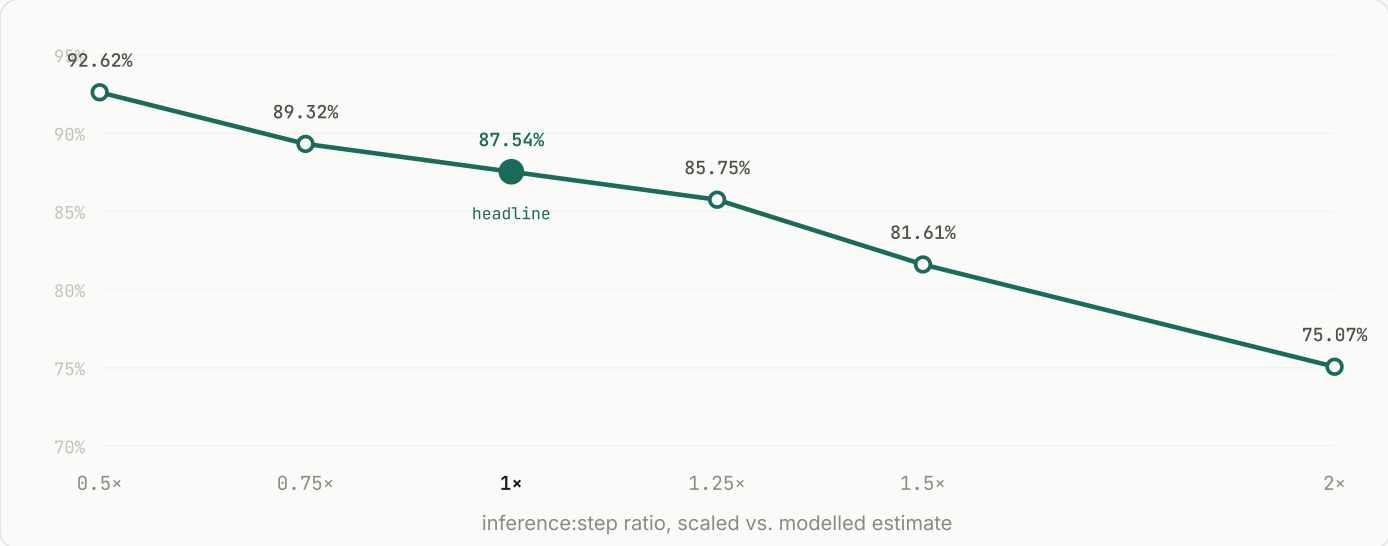
That last part is a useful sanity check. If no-history had turned out to be the biggest saving, it would mean we hadn't really given the baseline its caching. It comes out small *precisely because we did.*

07 Stress-testing the assumptions

Two sweeps over the assumptions most likely to draw pushback.

The inference:step ratio

The core assumption is that most agent steps are deterministic. Say we've got that wrong. To check, we scale every task's IW inference count from half our estimate up to double it, and re-run the whole portfolio.



Portfolio cost reduction as the inference:step ratio is scaled from 0.5× to 2×. Even if IW needs **twice** as many real inferences as we modelled, the run-cost still falls 75.07%. The result doesn't sit on a knife-edge.

The baseline's caching

Our headline gives the baseline the most generous caching there is, history included. In a lot of real deployments the history can't form a stable cacheable prefix, because retrieval changes the context on every step. When that happens the baseline gets *more* expensive and IW's lead grows.

BASELINE CACHING REGIME	COST REDUCTION	
History fully cached *(our headline, most generous to baseline)*	87.54%	floor
History not cached *(the more common real deployment)*	90.4%	likelier

We report the smaller of the two. The more realistic case is bigger.

08 What it means at scale

On a single task the gap is fractions of a cent, $0.15 versus $0.0187. But enterprises don't run one task, they run millions, and the difference scales straight up with volume. The model and its output quality stay exactly the same, because none of this changes the model, only how often and how heavily it gets called.

MONTHLY TASK VOLUME	BASELINE / MO	IW / MO	SAVED / YR
100K tasks / month	$15,009	$1,871	**$157,660**
1M tasks / month	$150,090	$18,707	**$1,576,596**
10M tasks / month	$1,500,905	$187,075	**$15,765,959**

Same model, same answers. The only thing that changed is where the orchestration runs.

09 The honest ledger

A paper that only lists upsides isn't worth much. Here's what IW costs, and where this analysis could be wrong.

WHAT IW SPENDS THAT THE BASELINE DOESN'T

- **Building the graph.** Engineers write the markdown network and the Python functions up front. That's a real cost, but a serious agent isn't free to build either (prompt engineering, eval harnesses, a retrieval pipeline), so the fair comparison is the *difference* between the two, not IW against zero.
- **Compute.** Every function call uses CPU. At cloud rates that's a fraction of a cent per task, three to four orders of magnitude less than the token cost it replaces.
- **Upfront design work.** IW asks the team to decide, while building, what's deterministic and what genuinely needs the model. That's more work than letting an agent sort it out at runtime. What you get back is the bill above, plus determinism and auditability an open agent loop can't really give you.

WHERE THIS COULD BE WRONG

- **The inference:step ratio is our modelling choice.** It's the assumption everything leans on. §07 shows the result holds from 0.5× to 2×. Below roughly a 0.35× inference share the two architectures start to converge. Worth knowing where that line is.
- **Token sizes come from documented ranges, not one customer's logs.** They're representative rather than measured on-site. Any real deployment should re-run the script with its own numbers, and it's built to take them.
- **This measures cost, not capability.** Some problems really do need an agent exploring an open-ended loop. Our argument is that most enterprise workflows aren't like that. They're known processes with a handful of reasoning points, and for those, paying agent prices is a choice rather than a requirement.

10 Reproduce it

Every number on this page came out of actually running the simulation. It's seeded, so you'll get the same figures on any machine. None of it is illustrative.

RUN IT YOURSELF

```
$ python3 iw-token-economics-sim.py # → iw-token-economics-data.json (every figure on this page) # → iw-token-economics-tasks.csv (all 1,000 tasks, per-row)
```

sim `iw-token-economics-sim.py`: pure standard-library Python, no dependencies. One cost function, three switches, Shapley decomposition, both sensitivity sweeps.

data `iw-token-economics-data.json`: the aggregates this page renders from, embedded below.

tasks `iw-token-economics-tasks.csv`: the full per-task table with parameters, both costs, and both reductions. Dig into it however you like.

seed `20260729`: change it and the per-task distribution moves a little, but the aggregate barely budges.

To test it against your own setup, swap in your provider's pricing, your own workload mix, and token sizes pulled from your traces. The headline result (that running orchestration in code beats running it in tokens) doesn't depend on the specific draws. It depends on the architecture.

INTELLIGENCE WAREHOUSE RESEARCH JULY 2026

Authors. Akhil Singh, Nidhi Prabhu, and Aniruddha Tammewar.

The claim, in one line. Move orchestration out of the prompt and into code, and enterprise AI processes 95.97% fewer tokens and runs at 87.54% lower cost, measured against a baseline we gave every caching advantage, across 1,000 tasks, and reproducible from a single seeded script. Same model, same answers, run in a different place.

Figures rendered live from `iw-token-economics-data.json`. Simulation seed 20260729 · n=1,000 · Claude Sonnet-class pricing.