

HASHIRU

WHITEPAPER • NOVEMBER 2026

Agents That Hold the Money

HASHIRU v2 gives teams of AI agents a budget to hire models and other agents, and keeps them within it

THE SYSTEM

Agents with their own budgets

Each agent gets a budget from the agent that hired it, and spends it hiring models and other agents.

THE GUARDS

Built-in spending limits

Every call is capped at what its agent has left, unspent money comes back when a hire is fired, and an agent on its last step must answer.

THE RESULT

Limits that hold

Across 551 live hires, none spent more than 1.11 times its budget. The first version let one spend 372 times.

Kunal Pai
UCLA
kunpai@ucla.edu

Harshil Patel
UC Davis
hpppatel@ucdavis.edu

HASHIRU v2 is source-available under the PolyForm Noncommercial License 1.0.0 at https://github.com/HASHIRU-AI/HASHIRU

Executive Summary

A team of AI agents spends money every time it uses a model or another agent, and a strong cloud model costs more than a cheap one. HASHIRU v2 gives this money to the agents themselves. Each agent gets a budget from the agent that hired it and spends it by hiring models and other agents (over A2A and MCP). HASHIRU's runtime, the hub, keeps this safe. It caps every call at what the agent has left, returns unspent money when a hire is fired, and makes an agent on its last step answer.

What HASHIRU v2 delivers

- **Limits that hold.** Across 551 hires in 120 live runs, no hire spent more than 1.11 times its budget (13 went over by 9 to 11%, because the hub undercounted tokens in long inputs). The first version let a hire spend 372 times its budget and lost half its runs.
- **Long inputs kept away from the CEO.** Not showing long documents to the CEO agent raised the score (the share of points answered right) from 0.463 to 0.599, a gain of 0.136 (95% interval +0.067 to +0.216).
- **Where the agents win, and what holds them back.** With enough budget to run the CEO, the agents beat the cheap model alone on hard jobs where the market has a specialist (79% against 69% right) and match it overall (0.737 against 0.740). But they give the expensive model too little money, so its 79 hires took 55% of the spending and got 12 answers right.

Where it stands

These are small experiments ($33.21 of API calls in total, plus about $24 of GPU time), and our results tables fail three of seven validity checks (§4.6). Re-scored at larger budgets, a strategy that adapts during the task climbs to 0.88 while the cheap model alone stays near 0.75, but we have run the agents live only at the smaller budget. A 9B open model trained on a solver's splits comes within 0.04 of the solver (0.750 against 0.788), but does not beat it (§5.3).

Contents

1 Introduction 2
2 Who Splits the Money Today 4
3 An Economy for Agents 5
4 How We Measured 10
5 Results 13
6 Limitations and Scope 20
7 Toward Economies of Agents 21

1

Introduction

Teams of AI agents cost money, and how a team splits its budget decides what it can do. In a hierarchical system, a CEO agent hires managers and workers, as in HASHIRU v1 [1] or the orchestrators of AutoGen [2], MetaGPT [3] and Magentic-One [4]. Every hire costs something, and prices differ a lot. A strong cloud model costs more than a cheap one, and an outside agent may charge for each exact answer it sells. A team that spends on the wrong steps fails the steps that needed the money, and a team that cannot limit its spending cannot be trusted with a budget at all.

Some agent software already hands money to the agents. Automaton's `fund_child` sends credits to a child agent [5], wayland-core's `delegate` gives a sub-agent a dollar limit [6], and AgentLance lets a manager model set budgets for subtasks [7]. Other systems leave the agents out and split the money with an optimizer instead (BAMAS [8], ZEBRA [9]). Benchmarks of budget awareness (BAGEN [10], R^3-Bench [11], EcoAgent-Bench [12]) measure one agent spending its own tokens or actions. Two things are missing. A limit that is checked after a call does not stop the call from going over it. And none of these systems compares the agents' split with the best split the money could buy. No system lets agents hold the money for a whole hiring tree, limits every call before it is made, and measures the result against a known best split.

To address these limitations, we built HASHIRU v2. Every agent runs as its own A2A service, every tool is served over MCP, and every hire takes its budget out of its parent's budget. The runtime (the *hub*), not the prompt, keeps spending safe. It caps each call at what the agent has left. It refuses a call that cannot pay for an answer and says what the call would cost. It returns unspent money when a hire is fired, and it makes an agent on its last step answer with what it has. Every reservation, call and refund is logged, so we can measure how well the agents split the money, and we measured it on a benchmark where the best split is known.

What this whitepaper contributes

- **HASHIRU v2, an economy for agents.** Each agent gets a budget from its parent, in dollars and, for local models, in memory. It can say how hard a job is and let the hub hire the cheapest model that fits, or buy from a priced remote agent or a whole remote team over A2A (§3).
- **Built-in spending limits.** Budgets are set aside when an agent is hired and refunded when it is fired. Every call is capped at what its agent has left before it is made (as accurately as the hub counts tokens), and an agent on its last step gets one final turn in which it must answer (§3.2).
- **Limits that hold, and a measured account of where agents win.** Across 551 live hires, none spent more than 1.11 times its budget, where the first version let one spend 372 times. Keeping long inputs away from the CEO raised the agents' score (the share of points answered right) by 0.136. With enough budget to run the CEO, the agents beat the cheap model alone on hard jobs that a specialist can do (79% against 69% right) and match it overall. Re-scoring at four budget levels shows where a strategy that adapts pulls ahead (§5).

The limits matter. Without them, the same agents spent up to 372 times a hire's budget and lost half their runs to an empty budget. With them, no hire went over by more than the hub's error in counting tokens. How the agents split the money is the open problem, and the logs turned it into something we could work on. They showed the CEO paying to reread every long document on every turn, and hiding those documents raised the score by 0.136. They also show where the agents win and why they do not win more. The agents send hard logic grids to an exact solver and retry hard jobs on more expensive models, which puts them ahead of the cheap model on hard jobs that a specialist can do. But they give the expensive model less money than it needs, so its hires took 55% of the spending and got 12 of 79 answers right. Re-scoring the same projects at other budgets shows what is at stake. As the budget grows eightfold (from a quarter of what the best answers cost to twice that), the cheap model alone stays near 0.75, while a strategy that adapts during the task, as the agents do, rises from 0.70 to 0.88. The remaining problems are just as specific. The expensive model is underfunded, a CEO that pays for every turn cannot work below a minimum budget, and the CEO answers some jobs itself instead of handing them off. Section 7 takes each one in turn.

A third round, on open models served on a rented GPU, asks whether a small model can learn to split the money. Trained on a solver's splits, a 9B model comes within 0.04 of the solver and scores 0.15 to 0.22 above the same model untrained (§5.3). It cannot beat the solver, which costs nothing to run.

2

Who Splits the Money Today

We group earlier work by who decides how the money is split, and when (Table 1).

An optimizer outside the agents. BAMAS [8] chooses the models with an integer program and how the agents work together with reinforcement learning, before the task starts. MasRouter [13] sends each query through a learned controller. ATOM [14] predicts how hard a query is and sets a budget for it, and MALBO [15] and SC-MAS [16] tune who is on the team. ZEBRA [9] splits dollars across the steps of a fixed pipeline with a solver before anything runs, and its language-model baseline loses to a knapsack solver. Mixture-of-Models [17] runs preset team profiles. In all of these, the agents doing the work do not hold the money. PURSE includes this setup as a baseline (a knapsack solver over measured success rates, §4.5).

Routing and cascades. Model routers pick one model per query. RouteLLM [18] learns from human preference data, Hybrid LLM [19] picks between a small and a large model by predicted difficulty, and FrugalGPT [20] tries models from cheap to expensive until an answer is accepted. RouterBench [21] scores routers on recorded outputs against the best possible router, the same method PURSE's replay uses. A router prices one query at a time. PURSE prices a project whose jobs share one budget, so money spent on one job is money another job cannot use.

Agents that fund agents. Agent runtimes released in 2026 let a model fund the agents it starts. Automaton sends credits to a child agent running the same runtime [5], and wayland-core's `delegate` tool gives a sub-agent a dollar limit out of what its parent has left, charged to the sub-agent and the agents below it [6]. AgentLance [7] lets a manager model set enforced budgets for subtasks, one level deep, with a market choosing the workers, and evaluates the whole system on benchmarks. HASHIRU v1 [1] came before these, with a CEO that hires and fires agents under cost and memory budgets, one level deep. None of them measures how well the agents divide the money.

Budget awareness in one agent. BATS [22] tells a tool-using agent how much budget it has left. BAGEN [10] finds that strong agents are not always budget-aware, and are too optimistic about tasks they cannot finish with the money left. R^3-Bench [11] has one model split its own output tokens or actions across six problems of hidden difficulty, and scores it against the best possible split, computed offline from how each problem responds to more effort. PURSE computes its best split the same way. EcoAgent-Bench [12] prices an agent's actions (lookups, searches, a stronger model, a human), AllocBench [23] measures investment in reusable tools, and CostBench [24] measures planning for the lowest cost and replanning when tools fail or prices change. In all of them, one agent spends on itself.

Local and cloud models, and agent economies. Minions [25] and PAPILLON [26] pair a local model with a cloud model under a fixed protocol. Virtual agent economies [27] and Magentic Marketplace [28] study markets between agents. Payment protocols (AP2 [29], the A2A x402 extension [30]) move the

money but do not decide how to split it. HASHIRU v2 uses the open protocols these build on, MCP for tools [31] and A2A for agents [32].

Table 1: Who splits the money, and when. Like the 2026 agent runtimes, HASHIRU v2 lets agents hold budgets. Unlike them, it caps each call before it is made and is compared with the best possible split.

Line of work	Who splits the money	When	Compared with the best split
BAMAS, ZEBRA, MasRouter	optimizer	before the task	no
RouteLLM, FrugalGPT	router, cascade	per query	best router
Automaton, wayland-core	the model	during	no
AgentLance	manager model	per subtask	no
R^3-Bench, EcoAgent	one agent, on itself	during	yes
HASHIRU v2	**the agents, to depth 2**	**during, each call capped**	**yes**

HASHIRU v2 shares agent-held budgets with these runtimes, and borrows R^3-Bench's way of computing the best split, but it differs in two ways. (1) *Limits before the call.* A hire sets money (and memory) aside that firing refunds, and each call's output is capped at what its agent has left before the call is made. wayland-core's sub-agent, by contrast, stops itself only after it has gone over its limit. (2) *Compared with the best split.* To our knowledge, it is the first system whose agent-held budgets, across a hiring tree, are compared with a solver, an up-front plan and fixed teams at the same budget.

3

An Economy for Agents

HASHIRU v2 rebuilds v1 [1] on open protocols. A user's request goes to a CEO agent. The CEO may hire *managers* (agents that use tools and can hire others) and *workers* (agents that answer once), up to a set depth (two levels below the CEO by default). Every agent runs as its own A2A service with its own Agent Card, and every tool is served over MCP. HASHIRU itself can be reached as an OpenAI-compatible model, an A2A agent or an MCP server, all from one process on one port. Every model call, tool call and message is written to a log (a JSONL trace). Figure 1 shows one hiring tree.

3.1 From v1 to v2

v1 had one level of hiring, a hand-written table of resource costs, and no study of when a local model beats a cloud one (limits it stated itself). v2 lets managers hire with money they were given. It prices every cloud call from LiteLLM's price list [33]. Local calls cost no dollars but take the memory their model really uses, measured from Ollama [34]. And it caps every call at what its agent has left. Rules that v1 left to prompts (who may spend what, and where private data may go) are now enforced in code, because in an earlier prototype, rules written only in prompts were broken.

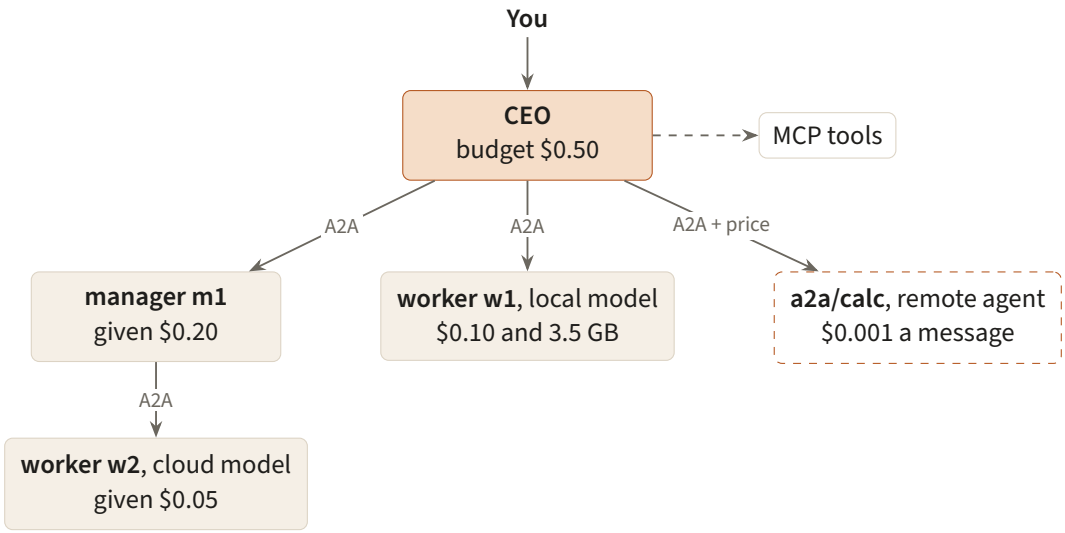


Figure 1: One hiring tree. Each hire takes its budget out of its parent's (and a local model also takes the memory it uses). Firing returns what the hire did not spend. A remote agent is paid its stated price out of the budget of the agent that hired it.

3.2 Built-in spending limits

Each agent a is given an amount R_a out of its parent's budget, and one rule says what it can still spend.

One rule for every budget

$$\text{left}(a) = R_a - \text{spent}(a) - \sum_{c \in \text{hires}(a)} \text{hold}(c) \tag{1}$$

A hire that is still working holds its whole amount. A fired hire holds only what it (and every agent below it) actually spent.

So firing returns the unspent part. Two hires running at the same time cannot spend the same dollars, because each holds its amount from the moment it is hired. Table 2 walks through one session.

Table 2: One session's budgets, in dollars. The CEO starts with $0.50.

Step	CEO left	m1 left
Session opens, `budget_usd` = 0.50	0.50	
CEO spends 0.01 thinking	0.49	
CEO hires manager m1 (0.20) and worker w1 (0.10)	0.19	0.20
m1 hires w2 (0.05) and spends 0.02	0.19	0.13
w1 spends 0.03, then the CEO fires it	0.26	0.13

A rule only works if it is enforced, and HASHIRU enforces this one in code on every call, not in the prompt. Three guards keep it true.

(1) *Reservations.* A hire cannot take more than its parent has left. A local hire must fit in free memory (or under a cap, `memory_mb`). A remote agent's stated price must fit in its budget before each message.

(2) *Capped calls.* Before a cloud call, the hub prices the input and caps the output at what the agent can still pay for, so the call cannot cost more than the agent has. A call that cannot pay for even a short answer is refused before it reaches the provider, and the refusal says what the call would cost, so the parent can give more.

(3) *A last turn.* An agent with one step left (its last allowed turn, or a budget that covers an answer but not another working turn) answers with what it has instead of ending with nothing.

Together, these keep each agent within its budget, up to the hub's error in estimating tokens, whatever the agent decides. Section 5 shows each one at work. Separately, `max_usd` caps real spending for the whole process. Because every reservation, call and firing is in the log, the accounts can be rebuilt from the log alone. Figure 2 follows one hire from its reservation to its refund.

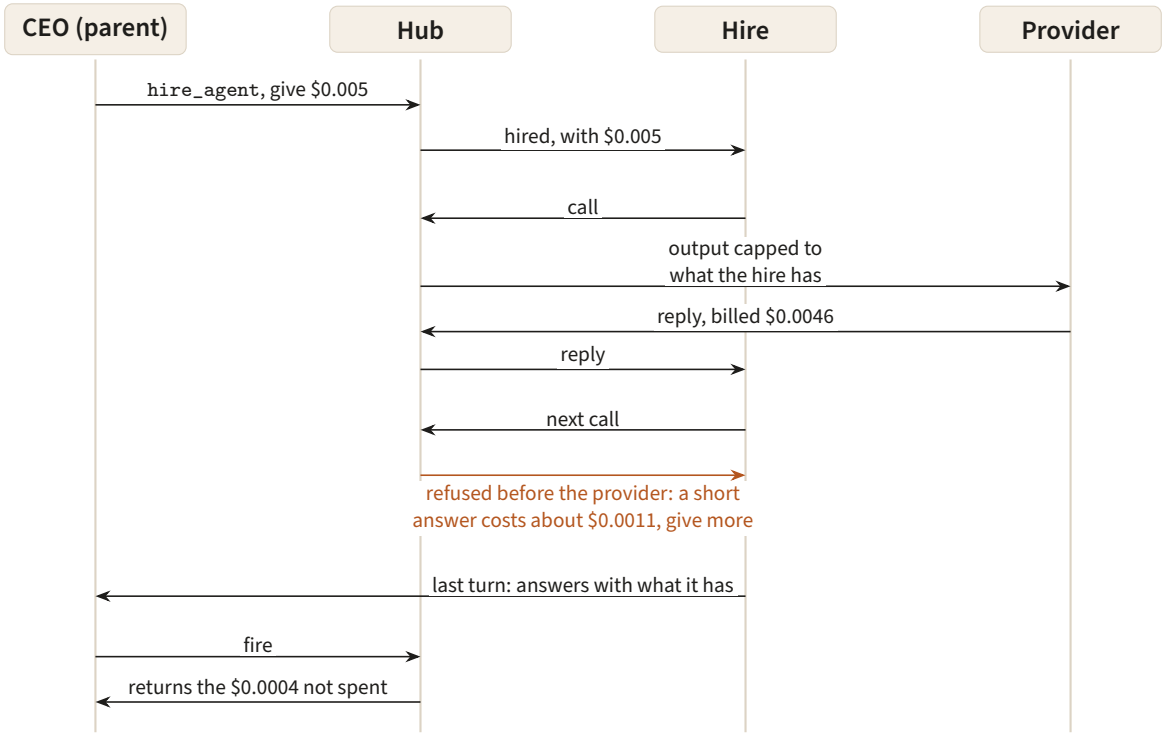


Figure 2: One hire's money, from reservation to refund, using the amounts of the median `muse-spark-1.3` hire in §5. The hub takes the hire's budget out of its parent's, caps each call's output at what the hire has, refuses a call that cannot pay for a short answer before it reaches the provider (and quotes the price), and returns what was not spent when the hire is fired.

3.3 Hiring by declared difficulty

An agent that hires may name a model, or say how hard a job is (easy, medium or hard) and let the hub choose. The hub turns a difficulty into a list of models, best first, and hires the first one that the settings, the parent's budget and the data's privacy allow. An easy job tries small local

models first, then cheap cloud models. A hard job tries the most expensive cloud models first. A worker that fails or gives an empty answer moves up its list and gets the message again, up to a limit (`max_escalations`, two by default). The list of models comes from the machine itself (Ollama models that are downloaded and can call tools, and every cloud model in LiteLLM's list whose provider key is set, ranked by price or size). The declared difficulty is the agent's own guess, so PURSE checks it against each job's hidden difficulty (§5).

An agent can instead name the kind of job (`kind`) and let the hub choose by measured success. Given a table of each model's success rate and mean cost for each kind of job (`success_rates`, which can come from the hub's own logs), the hub hires the cheapest model that succeeds at least 70% of the time on that kind (`route_min_rate`), then the others, most reliable first. The hiring tool lists the kinds the table knows, so the agent can name one. If the hire's budget cannot pay for one average attempt on the model the hub picks, the hub does not make the hire. It quotes what an attempt costs and lists the models the budget can pay for, so the agent can fund the hire in full or buy something cheaper before any money is spent. Hires in a proposed plan get the same check. We added this quote after the second round, and it has not yet been run live.

A model served outside the cloud providers, such as one on vLLM [35] on a rented GPU, gets its own server address (`endpoints`). It is priced per token from a prices file, like any cloud model.

3.4 Make or buy over open protocols

Besides models, a manager may hire a remote A2A agent as a worker (`a2a/<name>`). The hub reads the agent's card and prices it through a small A2A extension. The card states a price per message or per token, and each reply reports what the request cost. The hub charges the larger of the two to the hire's budget, and only talks to the agent at its configured URL. Since a HASHIRU hub is itself an A2A agent, one hub can hire another. A whole team, with its own CEO and budgets, then looks like one priced worker to the agent that hired it. This makes hiring a make-or-buy choice. An agent can make an answer with a model it pays per token, or buy it from an outside agent at a set price. Either way, it can attach parts of the user's request to its message word for word, so a long input reaches the hire in full. With `request_by_reference` on, the CEO never reads a long section at all. It sees each section of 2,000 characters or more cut down to its heading and first two lines, with a note on how to pass it on. A message to a hire that names a section by its key (`j3` for "### j3 (logic grids, ...)") carries the whole section. The CEO then pays to read the request's outline on every turn, instead of the whole request.

3.5 Quotes, approvals and privacy

Three more features read naturally in money terms. *Plan mode is a quote.* The CEO may only read and propose. The hub previews each proposed hire as it would be made (the model the router picks, the money it takes, the memory it uses) and checks the totals against the budget, and approving the plan approves its hires. An approved plan becomes a graph (LangGraph [36]) that hires, runs each step once its inputs are ready, fires each hire after its last step, and lets the CEO answer once. *Approvals are spending authority.* A hire or a turn above a set amount waits for the user. *Privacy limits where work may go.* Private data never reaches a cloud or remote model as it is. It is replaced with

placeholders on the way out and restored on the way back (Presidio's recognizers [37] and a local model find the values), or the call is refused.

3.6 Giving a team a budget

A team is one YAML file and one command (Figure 3). This file sets up the market from our evaluation, with three cloud models, a priced logic-grid solver over A2A, and the second round's measured success rates for level-5 arithmetic (Table 6). Every request then gets a CEO with a budget of `budget_usd`, the whole process stops at `max_usd`, and the log records every reservation, call and refund.

```
# team.yaml
ceo_model: gpt-6-luna
models: [gpt-5-nano, gpt-6-luna, meta/muse-spark-1.3]
budget_usd: 0.10              # the CEO's budget for each request
max_usd: 5.00                 # real spend, for the whole process
request_by_reference: true    # long sections stay out of the CEO's context
a2a_peers:
  csp: {url: http://127.0.0.1:9512/}    # a priced logic-grid solver, hired as a2a/csp
success_rates:                # kind of job -> model -> [success rate, mean cost in $]
  arithmetic level 5: {gpt-6-luna: [0.25, 0.0031], gpt-5-nano: [0.53, 0.0073],
                       meta/muse-spark-1.3: [0.92, 0.0961]}

$ uv run --env-file .env hashiru ask --config team.yaml "Answer the jobs below ..."
```

Figure 3: A team with a budget. The market from our evaluation in one config file, and the command that sends it a request. Every key is described in this section.

3.7 Implementation

HASHIRU v2 is 6,659 lines of Python 3.13, with exact versions pinned for its protocol and model libraries. Its 475 tests (and 178 more for the benchmark) run without API keys or Ollama. The guards in §3.2 came out of measurement. Each was written test-first after a benchmark run showed what happens without it (§5). So were routing by measured success, hiding long sections from the CEO, and four smaller features. `attach` lets an agent pass sections of the user's request to a hire word for word. A switch turns off the web tools, so no answer can be looked up during an evaluation. Reasoning models that reject temperature 0 are supported. And a call refused because of a provider's rate limit now waits and retries (for up to 104 seconds, without being billed) instead of failing.

4

How We Measured

To know whether agents spend well, we need tasks where the best split of the money is known. Our benchmark, PURSE, builds such tasks. Each project is a set of jobs that share one budget. Every job has an exact answer, and every job has been tried on every option in advance, so the best split for any budget can be computed. The score then depends on how the money is split, and the gap to the best split shows how much a strategy left on the table.

4.1 Jobs and projects

A project has eight jobs, two from each of four job types (Table 3), and one budget B. Each job has a visible level from 1 to 5 (its stated size), a hidden difficulty (low, mid or high), a value of 1, 2 or 4 points chosen independently of difficulty, a standard prompt and an exact checker. A strategy reads every job with its level and value and answers with one JSON object. The score is the share of points answered correctly. Jobs are generated from random seeds, so they cannot be looked up, and HASHIRU's web tools are turned off in every run.

Table 3: PURSE's four job types.

Job type	Set by the level	Set by the hidden difficulty	Can be bought from
Arithmetic	operators (2 to 22), digits (2 to 8)	multiplications, digit sizes (carries)	`calc` (levels 1 to 4)
Logic grids	houses, attributes	guesses a constraint solver must make (0, 1, 2+)	`csp`; a free checker
Aggregation	records (40 to 2,500)	fields in the condition (1, 2, or 2 plus near misses)	no one
String pipelines	operations (2 to 10), length	alphabet, operations that depend on position	no one

For the second round (§5), the hidden difficulties were spread further apart, so that knowing them would be worth more. In arithmetic, the hidden difficulty also changes the numbers' digits (one fewer, the same, or two more) and the share of multiplications (10%, 35% or 70%). In aggregation, it sets how many records match (a handful, about 1 in 20, or about 1 in 7) and triples the near misses. In strings, it halves, keeps or doubles the starting string. At high difficulty all but one operation depends on position, and at low difficulty the Caesar shift is dropped.

4.2 How much of the difficulty is visible

With probability ρ, a job's hidden difficulty is the one its visible level suggests (levels 1 and 2 low, 3 mid, 4 and 5 high). Otherwise it is random. At $\rho = 1$ the level tells the whole story, so a plan made up front with the right statistics loses nothing to a strategy that adapts. At $\rho = 0$, difficulty can only be

learned by trying. The value of knowing the hidden difficulty,

$$\mathrm{VOI} = \mathrm{score}(\text{oracle knowing the difficulty}) - \mathrm{score}(\text{oracle knowing only the levels}),$$

is the most that any adaptive strategy can gain over the best up-front plan, and we report it for every setting. Each strategy gets the statistics it would measure on projects with its own ρ, which are each difficulty's success rates weighted by $P(\text{difficulty} \mid \text{level}) = \rho\,[\text{difficulty matches the level}] + (1 - \rho)/3$.

4.3 The market

The options are models at different prices (a cheap and an expensive reasoning model in the first round, plus a third, `gpt-5-nano`, in the second) and outside agents reached over A2A. Two outside agents are exact programs. `calc` does exact arithmetic (it is HASHIRU's example agent, and reads expressions of up to 200 characters, so it refuses level 5), and `csp` solves logic grids. The second round adds `gridfast`, which sells the same grids at a quarter of `csp`'s price and gets 40% of them wrong (it swaps one attribute between two houses), so that the cheapest seller is not always a good one. Outside agents charge a price per message, tested at $c_{\mathrm{ref}} \times \{0.25, 1, 4, 16\}$, where c_{ref} is the cheap model's median cost on the jobs they can do. A free `grid_check` tool says whether a grid solution meets every clue, up to three times per puzzle (checking is cheaper than solving, as with unit tests).

4.4 Recorded results, replay and the oracle

Before any strategy runs, every job is sent with its standard prompt to every option that can do it (once in the first round, three times in the second), through a HASHIRU hub (one hire, one message, one fire). We call this *calibration*. The *results table* records each answer, whether it is right, and what it cost. A replayed call returns one of the job's recorded attempts, and each seed starts from a different one. The table has three uses. First, it gives the informed strategies their statistics, taken from the other half of the jobs, so a project never sees statistics that include its own jobs. Second, it defines the *oracle*. Oracle-full picks, for each job, the option that gets the most points under the budget, using each job's recorded success and cost (a multiple-choice knapsack, as in R^3-Bench [11]). No strategy can beat it. Third, it lets any strategy that sends standard prompts be scored by *replay* on recorded results instead of new model calls, as RouterBench does for routers [21]. Each project's budget is $B = \tau\, C_{\mathrm{full}}$, where C_{full} is what oracle-full spends at its best score when money is unlimited. So $\tau = 1$ is just enough money for the best answers, and $\tau = 0.5$ is half of that.

4.5 Strategies

The main comparison crosses who splits the money (a language model or a solver) with when (up front, or during the task). All four get the same project text, prices, statistics and budget (Table 4).

Around them are oracle-full, fixed strategies (every job on one model; a hybrid that picks by visible level, in the style of AgentCARD [38]; an equal split; a random choice), and HASHIRU variants (a CEO that names models, as in v1, and a CEO that sees prices but no statistics).

Table 4: The four strategies in the main comparison.

	Up front	During the task
LLM	**LLM split.** The CEO plans one hire per job in plan mode, and the plan runs unchanged (like ZEBRA's LLM baseline).	**HASHIRU.** The CEO hires, asks, fires, rehires and moves up to better models as results come in.
Solver	**Knapsack.** Picks options to get the most expected points under the budget, using the statistics.	**Cascade.** Takes the cheapest option expected to succeed, and moves up a tier when a check fails or two answers disagree.

4.6 Validity checks

Before any strategy runs, seven checks on the results table make sure PURSE really measures how the money is split (Table 5). We tested them on made-up tables built to pass, and on one table per check built to fail exactly that check. Check V8 (replay agrees with live runs) needs live runs.

Both tables behind §5 fail three of the seven, and not the same three. On the first table, V1 failed narrowly (the oracle beat the best fixed strategy by 9.7 points, not 10). V4 failed because the expensive model had room to improve in only one job type (long aggregation jobs), not two. V5 failed outright. Knowing each job's hidden difficulty was worth 0.009 of the points at $\rho = 0$, against the 0.08 the check asks for. We built the second table to fix these, with a third model, a cheap but unreliable grid seller, more spread-out difficulties and three attempts per job. V1 now holds (10.7 points). V5 still fails, and by more (0.002). The wider spread did separate the cheap models (`gpt-6-luna` answers 100% of low-difficulty aggregation jobs and 43% of mid ones). But knowing the hidden difficulty is worth between -0.007 and 0.002 of the points at every budget we tried (τ from 0.25 to 2), and 0.01 to 0.02 with the expensive model removed. With only four jobs per group, each difficulty's rates rest on twelve attempts, which are as noisy as the differences they are meant to show. V3 now fails because `gpt-5-nano` is not simply a weaker model. It beats `gpt-6-luna` on arithmetic (53% against 25% at level 5) and loses to it on aggregation and strings, so the three models cross over by job type instead of lining up, and on logic grids the expensive model never leads the cheap one by 20 points. V4 fails as before. So neither table can show what adapting during the task is worth, and the during-the-task column of Table 4 cannot be tested against the up-front one. Normally a failed check means a job generator is retuned and its jobs are measured again. We report both tables as they are, and a table that passes V5 is on the roadmap (§7).

Table 5: Validity checks on the results table, run before any strategy, and how the two tables behind §5 do.

Check	Holds when	First table	Second table
V1 Splitting matters	the oracle beats the best fixed strategy by ≥ 10 points at $\tau = 0.5$	**fails** (9.7)	holds (10.7)
V2 The budget is tight	the expensive model alone covers $<$ 60% of the points at $\tau = 0.5$	holds (59.6%)	holds (51.7%)
V3 Models separate	for each job type, a level where the cheap model succeeds $\geq 80\%$ (and a small model, if present, $\leq$ 40%), and one where the expensive model leads the cheap one by ≥ 20 points	holds (4 of 4 types)	**fails** (0 of 4)
V4 Room to improve	the expensive model succeeds $\leq 60\%$ at level 5 in $\geq$ 2 job types	**fails** (1 type)	**fails** (1 type)
V5 Hidden difficulty matters	VOI ≤ 0.02 at $\rho = 1$ and ≥ 0.08 at $\rho = 0$	**fails** (0.000; 0.009)	**fails** (0.000; 0.002)
V6 Buying responds to price	the oracle's buying falls ≥ 40 points across the price sweep	holds (53 points)	holds (56 points)
V7 No shortcut	the most common answer per job type scores $\leq 10\%$	holds (4.3%)	holds (5.6%)

5

Results

We ask three questions. Do agents that hold the money stay within it? Do they use the market that HASHIRU gives them? And how well do they split the money?

5.1 Setup

We ran two rounds. The first (2026-11-01) built the guards and measured them. The second (2026-11-02) added a third model and an unreliable seller, measured the models again on a wider set of jobs, and tested the agents on more projects. A third round (2026-11-03, §5.3) used open models on a rented GPU, with no live agent runs.

Models and prices. The CEO is `gpt-6-luna` ($0.10 / $0.50 per million input / output tokens). The first round's market has `gpt-6-luna` (the cheap model), `muse-spark-1.3` from Meta's Model API ($1.25 / $4.25, the expensive model), and the `calc` and `csp` agents. The second adds `gpt-5-nano` ($0.05 / $0.40) and `gridfast`. Prices are LiteLLM's list prices on the run dates. `gpt-5-nano` is the cheapest per token, but it reasons for longer, so one attempt on it costs three times as much as one on `gpt-6-luna` ($0.0030 against $0.0010). We tried other small models and dropped them (Gemma 4 31B, whose hosted endpoint did not answer, and Muse Glimmer 30B, which costs more per token than `gpt-6-luna`). We left out local models, since on the laptop we had (Apple M5, 24 GB) a 4B thinking model took six minutes on one arithmetic job.

Calibration. The first round sent 120 jobs (two per job type, level and difficulty) once to each option, for $0.18 on `gpt-6-luna` and $3.18 on `muse-spark-1.3`. The outside agents are our own programs and cost nothing real. The second round sent 240 jobs (four per group) three times to each option, for $0.70 on `gpt-6-luna`, $2.17 on `gpt-5-nano` and $20.32 on `muse-spark-1.3`. Table 6 shows where the second round's models differ, and where they cross over.

Table 6: Share of attempts answered right in the second round's calibration, by level (1 to 5; 36 attempts each).

Job type	`gpt-5-nano`	`gpt-6-luna`	`muse-spark-1.3`	Outside agent
Arithmetic	100 100 100 81 **53**	94 100 97 **64 25**	100 100 100 92 92	100 100 100 100 **0** (`calc`)
Logic grids	94 100 100 100 94	100 94 94 94 94	100 100 100 100 100	100 at every level (`csp`)
				67 67 50 58 50 (`gridfast`)
Aggregation	100 92 **44 25 0**	92 86 64 53 **36**	100 100 92 100 **47**	none
String pipelines	100 94 92 64 **47**	92 83 94 86 67	100 100 100 94 100	none

What these tables can show. Three of the seven validity checks fail on each table (§4.6), and each failure limits what this section can claim. V5 fails on both (knowing the hidden difficulty is worth at most 0.009 of the points at $\rho = 0$), so these runs cannot show what adapting during the task is worth, for any strategy. We compare the up-front and during-the-task strategies of Table 4 on what they score and spend, not on how well they adapt. V4 fails, so the expensive model solves nearly every job it is given (it drops to 60% or below only on level-5 aggregation). V3 fails on the second table, so its three models do not line up from weak to strong, since `gpt-5-nano` beats `gpt-6-luna` on arithmetic and loses to it on aggregation and strings.

Runs. The first round drew ten projects at $\rho = 0.5$ and $\tau = 0.5$ (mean budget $0.056), and ran HASHIRU v2 live twice on each of the first six (mean budget $0.043). Its outside agents charged $0.001 a message live, against the c_{ref} of $0.00023 that the replayed strategies paid. The second round draws twenty projects with the same ρ and τ (mean budget $0.058) and runs every strategy three times on each. HASHIRU v2 and the LLM split use three seeds, and the replayed strategies use three of each job's recorded attempts. Its outside agents charge c_{ref} ($0.000215 a message, and a quarter of that for `gridfast`), live and in replay alike. The solvers and fixed strategies are replayed on the results table. The LLM split makes its plan live and carries it out by replay. Each is scored under two budget rules. Under the first hub's rule, a call only needs some money left and is charged in full, even past the budget. Under v2's rule, a call is capped at what is left before it is made, as it is for HASHIRU v2. Replay cannot know what a capped call would have answered, so it counts it as no answer. In the first round's live runs, all ten calls by hires that hit their cap returned nothing.

We threw away two attempts at the second round's live runs. The first sent nine processes at once, went over the provider's limit of two million tokens a minute, and lost runs to refused calls. We made a rate-limited call wait and retry (for up to 104 seconds), and no call in the runs we report failed for this reason. In the second attempt, the project text put the market, the statistics and the answer format after the last job, inside that job's section, which `-by-reference` hides from the CEO. The text now puts them before the jobs, for every version. In total we spent $33.21. The first round cost $4.50. The second cost $23.42 to calibrate ($20.32 of it on `muse-spark-1.3`, and $0.23 on pilots), $4.30 for its live runs and $0.99 for the two discarded attempts.

5.2 What we found

Overall, our experiments show the following results:

The guards cap every hire. The second round made 120 live runs of HASHIRU v2 (two versions, twenty projects, three seeds) with 551 hires. No hire spent more than 1.11 times its budget. Thirteen went over, by 9 to 11%, each on a single call, and all thirteen were aggregation jobs, whose inputs are long lists of records. A call's output is capped at what the budget has left after the estimated cost of its input, so an overrun can only happen when the hub undercounts the input's tokens by more than its 10% margin. In the first round, none of 48 hires went over (Table 7).

Each guard is needed. In the first round we ran the same projects with HASHIRU as first built (without the call cap and the last turn), and with the call cap alone (Table 7). Without the cap, a budget was a limit checked before a call, and the call's cost was charged after it. Hires given $0.0005 spent up to 372 times that, the overruns were charged to the CEO's budget, and three of six runs ended with no answer after spending 153% of the budget. With the cap alone, no hire went over, but the CEO reread every job, long documents included, on every turn, and four of six runs ended without an answer (three because the CEO could not pay for its own answer, one at its 40-turn limit). With the last turn added, two of twelve runs were lost and the score rose to 0.566.

Table 7: HASHIRU with and without its guards, on the first round's six projects.

	As first built	Capped calls only	HASHIRU v2
Runs	6	6	12
Mean score	0.374	0.278	**0.566**
Hires that spent past their budget	9 of 26	0 of 25	**0 of 48**
Largest overrun (spent / given)	372×	1.0×	**1.0×**
Runs that ended without an answer	3 of 6	4 of 6	**2 of 12**
Spent / budget	1.53	0.62	**0.71**
Calls refused with a price quote	0	19	25

Keeping long inputs away from the CEO raises the agents' score by 0.136. The second round ran HASHIRU v2 as it was, and with two changes. Long sections were hidden from the CEO (`request_by_reference`), and hires were chosen by the success rates the CEO is shown (`success_rates`). With both, the agents scored 0.599 against 0.463, a difference of $+0.136$ over the twenty projects (95% interval $[+0.067, +0.216]$). The CEO reads the request's outline (8.7 thousand characters on the first project) instead of the whole request (264 thousand), and its own calls fell from 43% of the budget to 32%. Messages to outside agents now carry the puzzle the CEO names, so `csp` answered 48 of its 49 messages right, where before, 24 of its 44 replies had been "no grid puzzle in the text". The outside agents together answered 74 of 119 right, against 30 of 122. Runs where the CEO could not afford to read its request once fell from 12 to 0, and lost runs from 17 to 11. We did not test the two changes separately. Every effect above comes from hiding long sections, and what choosing by success rates adds on its own is not measured.

With enough budget to run the CEO, the agents match the cheap model alone. Table 8 scores every strategy under v2's budget rule, with the first rule's scores beside them. On the sixteen projects whose budget is $0.006 or more, the agents scored 0.737. That is about the same as `gpt-6-luna` alone (0.740; difference -0.003, interval $[-0.038, +0.030]$) and the equal split (0.761; -0.024, $[-0.056, +0.007]$),

and better than random choice ($+0.067$, $[+0.005, +0.134]$). The agents also scored above the cascade, the LLM split and `gpt-5-nano` alone, but by margins too small for sixteen projects to settle. They trail the knapsack (-0.080, $[-0.129, -0.023]$), and they spent 3.9 times what the cheap model alone spent.

The agents win on hard jobs that a specialist can do. Two job types have an outside agent in the market (`calc` for arithmetic, `csp` and `gridfast` for logic grids). On the sixteen projects, over the hard jobs of those two types (26 jobs, three runs each), the agents answered 62 of 78 right (79%), against 54 of 78 for `gpt-6-luna` alone (69%). That is 24 against 18 on hard arithmetic, and 38 against 36 on hard grids. They were ahead on six projects and behind on two, and the interval over projects only just reaches zero. They send grids to `csp`, which answered all 41 of its messages on these projects right, and they give hard jobs a second hire more often than easy ones (46 of the 81 jobs that went to more than one hire were hard, while hard jobs are 37% of all jobs). On the other jobs they do worse than the cheap model (226 of 306 right, 74%, against 239, 78%), and the two cancel out to the matching overall score above.

Below a minimum budget, the agents cannot work. Four of the twenty projects have budgets of $0.0009 to $0.0051, because their jobs are cheap to solve and $B = \tau\, C_{\text{full}}$. On them, `gpt-6-luna` alone scores 0.724, and the LLM split, whose one planning call comes out of the budget, scores 0.635. The agents score 0.046. A CEO that hires, asks and fires pays for a turn at each step. All eleven runs the agents lost were on these four projects, where the CEO came back empty after spending most or all of the budget on its own calls. On the other sixteen they lost none of 48. Over all twenty projects, the agents' 0.599 is below the cheap model alone (0.736).

Table 8: Strategies on the second round's twenty projects ($\rho = 0.5$, $\tau = 0.5$, mean budget $0.058), sorted by score under v2's budget rule, and on the sixteen projects whose budget is $0.006 or more. Each runs three times per project (sixty runs). "Vs. Luna" is spending divided by that of every job on `gpt-6-luna` ($0.0079 a project). The last column replays the same strategies under the first hub's rule, which charges a call in full even past the budget. HASHIRU v2's two versions run live, as it was and with long sections hidden (both during the task).

Strategy	Score	Sixteen	Spent ($)	Of budget	Vs. Luna	Own calls ($)	First rule
Oracle-full (knows outcomes)	0.826	0.838	0.028	0.48	3.50×	0	0.830 (0.49)
Knapsack (solver, up front)	0.788	0.817	0.035	0.60	4.42×	0	0.794 (0.68)
Every job on `gpt-6-luna`	0.736	0.740	0.008	0.14	1.00×	0	0.723 (0.15)
Equal split	0.713	0.761	0.011	0.18	1.35×	0	0.713 (0.18)
LLM split (LLM, up front)	0.695	0.709	0.033	0.57	4.15×	0.006	0.699 (0.69)
Cascade (solver, during)	0.682	0.712	0.050	0.86	6.29×	0	0.762 (1.58)
Every job on `gpt-5-nano`	0.628	0.700	0.023	0.39	2.87×	0	0.630 (0.41)
Random	0.612	0.670	0.029	0.50	3.69×	0	0.614 (1.05)
HASHIRU v2, long sections hidden	**0.599**	**0.737**	**0.031**	**0.52**	**3.84**×	0.006	live
Hybrid by level	0.493	0.494	0.052	0.90	6.60×	0	0.531 (2.14)
HASHIRU v2 as it was	0.463	0.579	0.036	0.61	4.47×	0.025	live
Every job on `muse-spark-1.3`	0.342	0.427	0.054	0.93	6.82×	0	0.447 (2.04)

First rule: score, with spent / budget in parentheses. "Sixteen": score on the sixteen projects whose budget is $0.006 or more.

The agents respond to prices. A hire's call that its budget cannot pay for is refused before it reaches the provider, and the refusal quotes the price. In the second round, 203 calls were refused this way in

the version that hides long sections (253 in the other). In 112 of them, a later hire in the same run was given more than the refused one (156 in the other). Table 9 follows one such exchange from a first-round log. The CEO called a level-4 logic grid easy and gave it to `gpt-6-luna` with $0.001. The hire's first call reasoned up to the cap its budget allowed and returned nothing. Its next call was refused, and so was the same call on the expensive model after the hub moved it up. The CEO then gave five times as much to a new hire, and twelve times as much to a third, whose answer was right and worth 4 of the project's 22 points.

Table 9: A refusal with a price quote, and what followed, from the log of one first-round live run (project p2, second run; job j4, a level-4 logic grid worth 4 of the project's 22 points).

	Given	What happened
1	$0.001	The CEO declares the job easy, and the hub hires `grid4` on `gpt-6-luna`.
2		`grid4`'s first call reasons up to the cap its budget allows, spends $0.0010 and returns nothing.
3		Its next call is refused before it reaches the provider.
4		The hub moves `grid4` up to `muse-spark-1.3`, and the refusal quotes the price: "grid4's pool has $0.0000, and a call on meta/muse-spark-1.3 costs about $0.0005 to read its messages and $0.0011 for a short answer: reserve more."
5	$0.005	The CEO checks its own budget and hires `grid4b`, which answers that the clues are inconsistent (wrong).
6	$0.012	The CEO hires `solve4`, whose solution is right.

The agents choose the expensive model for the right jobs, but give it too little money. On the sixteen projects, 65% of the agents' tries on `muse-spark-1.3` (68 of 104) were on hard jobs, and half came after a cheaper model had tried the same job. Its 79 hires took 55% of what the agents spent ($0.99 of $1.81) and returned 12 right answers, while its measured success rate on hard jobs is 87 to 100%. The median hire was given $0.005, but an attempt on the expensive model costs a median of $0.006 to $0.025 depending on the job type (a mean of $0.010 to $0.061). So 78 of the 79 hires had a call refused because their budget could not pay for it, after spending nearly all of it first (a median of $0.0046). The price quote comes after the money is gone. And at $\tau = 0.5$, the budget pays for a full attempt on the expensive model on only a few jobs, so these hires buy calls that cannot finish.

Splitting the money pays off once the budget has room, and the cheap model alone cannot use it. The live runs use $\tau = 0.5$. Replaying the results table at τ of 0.25, 1 and 2 (no new model calls; Figure 4 and Table 10) shows where a strategy can win. On the sixteen projects, `gpt-6-luna` alone scores 0.740 to 0.749 at every budget. It spends 31% of the budget at $\tau = 0.25$ and 7% at $\tau = 2$, and has nothing to buy with the rest. The cascade adapts during the task, as the agents do. It tries the cheapest option expected to be good enough, moves up when a grid's check fails, and asks a third option when the two cheapest disagree. It trails the cheap model by 0.051 at $\tau = 0.25$ and leads it by 0.133 at $\tau = 2$ ($[+0.078, +0.192]$, ahead on 12 of 16 projects). The knapsack's lead grows from 0.029 to 0.162. The best single model also changes, since the expensive model alone passes the cheap one at $\tau = 2$ (0.815 against 0.749). The knapsack, which plans up front, leads by more than the cascade at every budget, so the sweep does not show that adapting beats planning (V5 fails, as above). What it shows is that a budget with room rewards a strategy that uses it. The agents ran only at $\tau = 0.5$, where the cascade still trails the cheap model and the agents' tries on the expensive

model are underfunded. We did not run them at a larger budget, so whether they follow the cascade there is not measured, but the sweep says that a larger budget is where they should win.

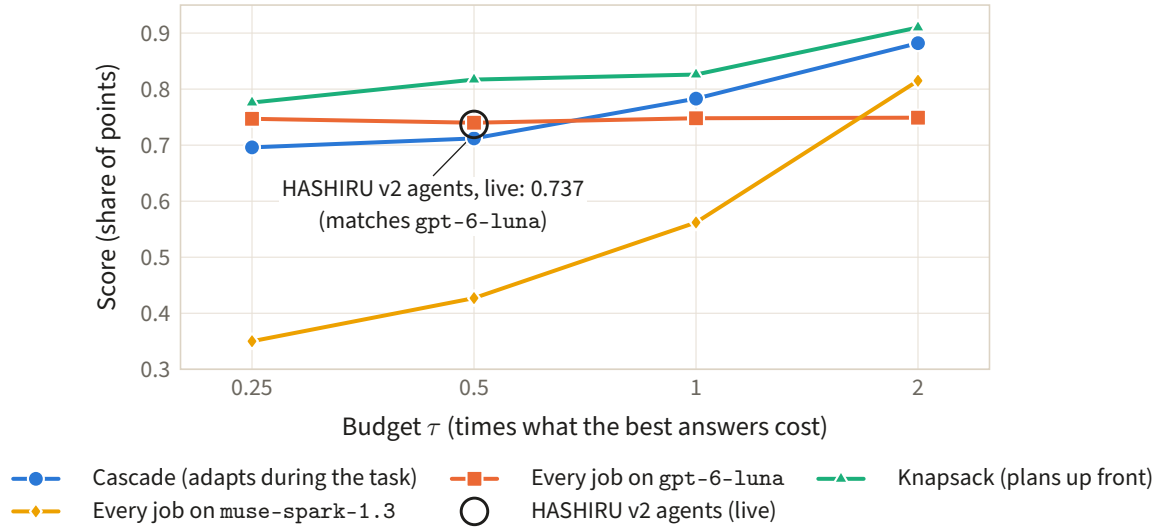


Figure 4: Where splitting the money pays off. Scores on the sixteen projects as the budget grows eightfold, replayed from the results table (three runs per project). The cheap model alone stays flat, while the cascade and the knapsack turn the extra budget into points. The agents ran live at $\tau = 0.5$ only (ringed). Table 10 gives the numbers and intervals.

Table 10: Replayed strategies on the sixteen projects at four budgets ($B = \tau\, C_{\text{full}}$, a mean of \$0.036, \$0.072, \$0.144 and \$0.288 a project), three runs each under v2's budget rule. The agents ran live at $\tau = 0.5$ only. The last row is the cascade's lead over the cheap model alone, with its 95% interval over projects.

Strategy	$\tau = 0.25$	$\tau = 0.5$	$\tau = 1$	$\tau = 2$
Every job on `gpt-6-luna`	0.747	0.740	0.748	0.749
Every job on `gpt-5-nano`	0.635	0.700	0.721	0.723
Every job on `muse-spark-1.3`	0.350	0.427	0.562	0.815
Equal split	0.748	0.761	0.791	0.808
Cascade (solver, during)	0.696	0.712	0.783	**0.882**
Knapsack (solver, up front)	0.776	0.817	0.826	0.910
Oracle-full (knows outcomes)	0.825	0.838	0.844	0.947
HASHIRU v2 agents (live)		0.737		
Cascade minus `gpt-6-luna`	-0.051 $[-0.108, +0.004]$	-0.028 $[-0.088, +0.031]$	$+0.035$ $[-0.036, +0.100]$	$\mathbf{+0.133}$ $[+0.078, +0.192]$

What the agents still get wrong. Besides the expensive model, two things separate them from the knapsack. First, the CEO keeps some jobs for itself and answers them worse than a fresh call would. In the version that hides long sections, it answered 169 jobs itself and got 85% right, while `gpt-6-luna` given each job on its own gets 91%. The 223 jobs it handed to hires were right 67% of the time, against 68%. Second, the difficulty the CEO declares barely matches the jobs' hidden difficulty (rank correlation 0.10 over 128 hires), and its own calls still take 32% of the budget. It rarely buys from the cheap, unreliable `gridfast` (seven messages, five of them right).

Holding the other strategies to v2's rule lowers the ones that overspent under the first rule. The cascade, the hybrid by level and the expensive model alone, which spent 1.58 to 2.14 times their budget under the first rule, fell by 0.080, 0.038 and 0.105. Random choice and the knapsack fell by only 0.002 and 0.006. It raised the cheap model alone by 0.013, because refusing a call the budget cannot pay for saves the money for later jobs (on one small project, the first rule let the cheap model's calls spend up to 2.9 times the budget, and it scored 0.4 to 0.6 less).

5.3 A third round on open models

Both results tables fail V5. Our cloud models are strong enough that a job's hidden difficulty rarely changes their answer, so knowing it is worth almost nothing. The third round asked two questions with open models. Do weaker models make the hidden difficulty worth knowing? And can a small model be trained to split the money the way the knapsack does? We served the models ourselves with vLLM [35] on one rented H100 GPU, for about 5.5 hours and about $24 of Lambda credits. No agent ran live in this round, and it made no paid API calls. Each open model is charged OpenRouter's list price (read on 2026-11-02), as if it were bought from a hosted provider. Each attempt's output is capped at 8,192 tokens.

Weaker models make the hidden difficulty visible, but not worth knowing. We measured three open models on 144 jobs (levels 1, 3 and 5, four per group), three attempts each. They are Ministral 3 3B ($0.10 / $0.10 per million input / output tokens), Qwen3.5 9B ($0.17 / $0.25) and Gemma 4 31B ($0.20 / $0.40). The hidden difficulty now moves success a lot. Qwen3.5 9B answers every level-3 arithmetic job of low hidden difficulty and none of the mid or high ones. But knowing it is still worth almost nothing. It is worth 0.006 of the points at $\rho = 0$, and -0.020 when the success rates are measured on half the jobs and used on the other half. The prices explain why. Gemma 4 31B is the strongest model (65% of its attempts right), and cheap per attempt (a mean of $0.0028). Qwen3.5 9B thinks until the cap on 294 of its 432 attempts, so an attempt on it costs twice as much ($0.0060) and is right 52% of the time. Ministral 3 3B is right 9% of the time. The best plan sends nearly every job to Gemma 4 31B, whatever its hidden difficulty. Five of the seven checks fail (V1, V2, V3, V5 and V6), so we did not build a results table on these models.

A small model trained on the knapsack's splits nearly matches it, but does not beat it. ZEBRA's language-model baseline loses to a knapsack solver [9], and so does our LLM split (Table 8). We asked how much of that gap training closes. Our splitter reads a project's outline in one prompt and replies with one option, or "skip", for each job. The outline holds the budget, the market, the measured success rates and costs, and each job's heading with the first 160 characters of its text. The reply is paid from the budget first, at Qwen3.5 9B's list price. We trained Qwen3.5 9B with LoRA [39] (rank 16, one epoch, 33 minutes) on 3,000 projects drawn from the second round's jobs, none of them in the twenty test projects. The target is the knapsack's split of what the budget has left after the reply. Table 11 scores the splitter on the second round's twenty projects at four budgets.

Training adds 0.15 to 0.22 at every budget, and the trained model picks the knapsack's option for 92% of jobs (the untrained one for 26%). It comes within 0.013 to 0.024 of a perfect copy. It still trails the knapsack by 0.023 to 0.052, because its reply costs about 8% of the budget and it can at best copy the solver. At $\tau = 0.5$ it scores 0.056 above the LLM split, but the interval includes zero ($[-0.031, +0.131]$). With thinking on, the untrained model never finished thinking within 8,192 tokens on any of the 80

Table 11: Splitting the money in one reply, on the second round's twenty projects at four budgets (three replays each, under v2's budget rule). Every model pays for its reply out of the budget. The perfect copy writes the knapsack's split exactly. It pays for its reply from an estimate of the prompt's length that is about half the real length, so its row is slightly high. The last rows give differences, with 95% intervals over projects.

Splitter	$\tau = 0.25$	$\tau = 0.5$	$\tau = 1$	$\tau = 2$
Knapsack (solver, free)	0.715	0.788	0.842	0.902
A perfect copy of the knapsack	0.687	0.766	0.833	0.900
Qwen3.5 9B, trained	**0.663**	**0.750**	**0.820**	**0.879**
Qwen3.5 9B, untrained, thinking off	0.513	0.578	0.660	0.661
Qwen3.5 9B, untrained, thinking on	0.000	0.000	0.000	0.000
LLM split (`gpt-6-luna`, Table 8)		0.695		
Trained minus untrained (thinking off)	$+0.150$	$+0.172$	$+0.159$	$+0.218$
	$[+0.077, +0.225]$	$[+0.096, +0.245]$	$[+0.085, +0.238]$	$[+0.132, +0.306]$
Trained minus knapsack	-0.052	-0.038	-0.023	-0.023
	$[-0.108, -0.008]$	$[-0.085, -0.001]$	$[-0.046, -0.003]$	$[-0.050, -0.004]$

prompts, so it answered nothing and still spent 38.5% of the budget. So training closes most of the gap between an LLM's split and a solver's, but a model that copies a free solver cannot beat it. The knapsack reads no job's text. A trained splitter could add something only where the text tells it what the statistics do not, such as a job's hidden difficulty, and V5 says that is worth almost nothing on these tables. The trained model is public (§7).

6

Limitations and Scope

Scale of the runs. The second round has twenty projects, three runs of each strategy on each, and three calibration attempts per job and option, so replay carries some of the models' own randomness. Our intervals are over projects, and with twenty (or sixteen) projects they are still wide. Both results tables fail three of their seven validity checks (§4.6), V5 among them, so neither round can show what adapting during the task is worth.

Live against replay. The other strategies are replayed with each job's standard prompt, while HASHIRU v2 ran live and wrote its own messages to its hires. Replay counts a call cut off at its budget as no answer, and the first round's live runs agree (all ten such calls by hires returned nothing). In the first round, the outside agents charged HASHIRU $0.001 a message live, about four times the c_{ref} the replayed strategies paid. In the second, live and replay paid the same price. The budget sweep (Table 10) is replay only, since the agents ran live at one budget, so it shows where a strategy can win, not that the agents do.

What was changed. The second round changed two things at once (hiding long sections, and choosing hires by measured success rates) and did not test them separately. The logs tie each measured effect

to hiding sections, but what success-rate routing adds on its own is not measured. Its rates are the same statistics the CEO is shown, so it cannot know more than the CEO is told.

Small budgets. Budgets follow $B = \tau\, C_{\text{full}}$, so a project of cheap jobs gets a small budget, and four of the second round's twenty have less than $0.006. A strategy that pays for its own turns cannot work below that, and we report those projects separately rather than drop them.

The open-model round. The third round ran no agents live, so it says nothing about how the agents spend on open models. Its jobs come four per group (two in each half), so the value of the hidden difficulty measured across halves is noisy, and seven of Gemma 4 31B's 432 attempts timed out and are missing. The 8,192-token cap shapes Qwen3.5 9B's cost, since most of its attempts run to it. The open models are charged list prices, not what the GPU cost. The trained splitter was trained and tested on the same job generators and the same market, so other options or prices would likely need new training.

The market and the system. Prices are list prices on the run dates. The live runs use three cloud models and no local model, so memory as a second currency is built into HASHIRU but not measured here. The declared difficulty is the agent's own guess. A call is capped using the hub's token count, which ran low by up to 11% of a budget on long aggregation inputs (13 of 551 hires). All agents in a hub share one API key, so the hub, not the provider, is what keeps them apart.

7

Toward Economies of Agents

HASHIRU v2 shows that AI agents can be given money and held to it. Each agent gets a budget from its parent, every call is capped at what its agent has left, firing returns what was not spent, and an agent on its last step must answer. Over 551 hires in 120 live runs, no hire spent more than 1.11 times its budget, where HASHIRU as first built let one spend 372 times. Keeping long inputs away from the CEO raised the agents' score by 0.136. On projects with enough budget to run the CEO, the agents beat the cheap model alone on hard jobs that a specialist can do (79% against 69% right) and match it overall (0.737 against 0.740), at 3.9 times its spend. A 9B open model trained on the knapsack's splits comes within 0.04 of the knapsack (0.750 against 0.788 at the live runs' budget), but a model that copies a free solver cannot beat it. Because every reservation, call and refund is in the log, the spending can be checked, measured and improved. That is how we built the guards and both changes, and how we found what is left to fix.

When to use HASHIRU v2 today

Our runs support five rules of thumb, each limited to the market we measured.

- **A hard spending limit.** Use HASHIRU when a team must stay within its money. Every hire stayed within 1.11 times its budget, while the same agents without the call cap overspent one by 372 times.
- **Hard jobs that a specialist can do.** Hire through HASHIRU when the market sells exact answers. On hard arithmetic and logic grids, the agents were right 79% of the time, against 69% for the cheap model alone.
- **A budget too small for the expensive model.** Expect the cheap model's score at a higher cost. At the budgets we ran, the agents matched `gpt-6-luna` alone (0.737 against 0.740) at 3.9 times its spend.
- **A very small budget.** Below about $0.006 a project, send each job to one cheap model. A CEO that pays for each turn used up the budget before it answered (0.046 against 0.724).
- **A budget with room.** Split the money instead of using one model. In replay, the cheap model alone stayed near 0.75 as the budget grew eightfold while the cascade reached 0.88. We have not yet run the agents at these budgets.

This is a step toward agent economies, where teams of agents hold budgets, buy from each other and account for every dollar. The pieces are already in HASHIRU. Priced remote agents and whole remote teams are hired like any model, prices travel in an A2A extension, and payment protocols (AP2 [29], the A2A x402 extension [30]) can settle what the accounts record. An organization could give a team of agents a budget the way it gives one to a department, and get back an answer, spending within that budget, and a log of where each dollar went.

Roadmap. Each remaining gap has a concrete next step, listed by payoff per dollar. The agents pick the expensive model for hard jobs but give it too little money, so more than half of what they spent went to its hires, and only 12 of 79 answered right. The hub already knows what an attempt on each model costs, from the same statistics it routes on, and it now quotes that price when the hire is made, instead of after the money is spent. The CEO can then fund the hire in full or buy something cheaper. This quote is built and tested but has not been run live, so it comes first. With that in place, the agents should be run at larger budgets, where the replays show a strategy pulling ahead of the cheap model (0.133 ahead at twice the cost of the best answers). The agents cannot work below a minimum budget, since a CEO that hires, asks and fires pays for a turn at each step. Below it, the hub can plan up front in one call, as the LLM split does, or answer cheap projects with one model. The CEO answers the jobs it keeps worse than a fresh call (85% against 91%), which argues for handing jobs off by default. The call cap is only as tight as the hub's token count, which can be checked against billed usage in the log, and its margin learned from it. The two changes of the second round should be tested separately. And a results table that passes V5 needs more than four jobs per group, since a difficulty's rate from twelve attempts is as noisy as the differences it is meant to show. Without it, the value of adapting during the task cannot be measured. Weaker open models make the hidden difficulty visible (§5.3), but their market failed because its strongest model was also cheap per attempt, so the next table needs a market whose prices rise with quality. Only on such a table could a trained splitter

add something a solver lacks. Finally, the market can grow to include local models, with memory as a second currency, which HASHIRU already supports.

Availability. HASHIRU v2 is source-available under the PolyForm Noncommercial License 1.0.0 at https://github.com/HASHIRU-AI/HASHIRU. The benchmark and the run data are available from the authors on request. The trained splitter (merged weights and its LoRA adapter) is at https://huggingface.co/guineapig/purse-splitter-qwen3.5-9b. HASHIRU v1's code is preserved at tag `v1.0-paper`.

Acknowledgements

We thank Parth Shah for building much of HASHIRU v1, on which this work builds. We thank Hugging Face, Groq and Lambda Labs for sponsoring HASHIRU and providing the resources for its development, and Y Combinator's Startup School for API credits.

References

[1] Kunal Pai, Parth Shah, and Harshil Patel. HASHIRU: Hierarchical agent system for hybrid intelligent resource utilization, 2025. URL https://arxiv.org/abs/2506.04255.

[2] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=BAakY1hNKS.

[3] Sirui Hong, Mingchen Zhuge, Jonathan Chen, Xiawu Zheng, Yuheng Cheng, Jinlin Wang, Ceyao Zhang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VtmBAGCN7o.

[4] Adam Fourney, Gagan Bansal, Hussein Mozannar, Cheng Tan, Eduardo Salinas, Erkang Zhu, Friederike Niedtner, Grace Proebsting, Griffin Bassman, Jack Gerrits, Jacob Alber, Peter Chang, Ricky Loynd, Robert West, Victor Dibia, Ahmed Awadallah, Ece Kamar, Rafah Hosn, and Saleema Amershi. Magentic-One: A generalist multi-agent system for solving complex tasks, 2024. URL https://arxiv.org/abs/2411.04468.

[5] Conway Research. Automaton. https://github.com/Conway-Research/automaton, 2026. Agent runtime; the `fund_child` tool transfers credits to a child agent. Accessed 2026-11-01.

[6] FerroxLabs. wayland-core. https://github.com/FerroxLabs/wayland-core, 2026. Agent runtime; the `delegate` tool gives a sub-agent a USD ceiling. Accessed 2026-11-01.

[7] Xiao Liu, Haoyang Li, Songwei Li, Hongbo Fang, Fengli Xu, Feng Shi, and James Evans. Markets, not planners: Decentralized orchestration of llm agents with private information, 2026. URL https://arxiv.org/abs/2608.23867.

[8] Liming Yang, Junyu Luo, Xuanzhe Liu, Yiling Lou, and Zhenpeng Chen. BAMAS: Structuring budget-aware multi-agent systems. *Proceedings of the AAAI Conference on Artificial Intelligence*, 40(35):29802–29810, 2026. doi: $10.1609/\mathrm{aaai.v40i35.40226}$. URL https://doi.org/10.1609/aaai.v40i35.40226.

[9] May Hamri and Inbal Talgam-Cohen. ZEBRA: Zero-shot budgeted resource allocation for LLM orchestration, 2026. URL https://arxiv.org/abs/2605.20485.

[10] Yuxiang Lin, Zihan Wang, Mengyang Liu, Yuxuan Shan, Longju Bai, Junyao Zhang, Xing Jin, Boshan Chen, Jinyan Su, Xingyao Wang, Jiaxin Pei, and Manling Li. BAGEN: Are LLM agents budget-aware?, 2026. URL https://arxiv.org/abs/2606.00198.

[11] Peisong Wang, Zhiwei Ma, Bowen Liu, Feixue Liu, Aochuan Chen, Chenyi Zi, Hongchuan Zeng, Yuhan Li, and Jia Li. r^3-bench: Llms struggle with resource-rational reasoning under shared budgets, 2026. URL https://arxiv.org/abs/2608.16033.

[12] Jie Wu, Ming Gong, Feixiang Cheng, and Qinqin Zhao. Ecoagent-bench: Evaluating economic decision-making in budget-constrained llm agents, 2026. URL https://arxiv.org/abs/2608.05519.

[13] Yanwei Yue, Guibin Zhang, Boyang Liu, Guancheng Wan, Kun Wang, Dawei Cheng, and Yiyan Qi. MasRouter: Learning to route LLMs for multi-agent systems. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 15549–15572, Vienna, Austria, 2025. Association for Computational Linguistics. doi: $10.18653/\mathrm{v1}/2025.\mathrm{acl\text{-}long}.757$. URL https://aclanthology.org/2025.acl-long.757/.

[14] Xinkui Zhao, Sai Liu, Yifan Zhang, Qingyu Ma, Zewen Lin, Naibo Wang, Guanjie Cheng, Chang Liu, and Yueshen Xu. ATOM: Instantiating budget-controllable multi-agent collaboration via nucleus-electron hierarchy, 2026. URL https://arxiv.org/abs/2605.26178.

[15] Antonio Sabbatella. MALBO: Optimizing LLM-based multi-agent teams via multi-objective Bayesian optimization, 2025. URL https://arxiv.org/abs/2511.11788. Master's thesis, University of Milano-Bicocca.

[16] Di Zhao, Longhui Ma, Siwei Wang, Miao Wang, and Yi Kong. SC-MAS: Constructing cost-efficient multi-agent systems with edge-level heterogeneous collaboration, 2026. URL https://arxiv.org/abs/2601.09434.

[17] Tims Pecerskis and Aivars Smirnovs. Mixture-of-Models: Unifying heterogeneous agents via N-way self-evaluating deliberation, 2026. URL https://arxiv.org/abs/2601.16863.

[18] 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 from preference data. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=8sSqNntaMr.

[19] 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 *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=02f3mUtqnM.

[20] 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. ISSN 2835-8856. URL https://openreview.net/forum?id=cSimKw5p6R.

[21] 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, 2024. URL https://arxiv.org/abs/2403.12031.

[22] Tengxiao Liu, Zifeng Wang, Jin Miao, I-Hung Hsu, Jun Yan, Jiefeng Chen, Rujun Han, Fangyuan Xu, Yanfei Chen, Ke Jiang, Samira Daruki, Yi Liang, William Yang Wang, Tomas Pfister, and Chen-Yu Lee. Budget-aware tool use enables effective agent scaling. In *Third Conference on Language Modeling*, 2026. URL https://openreview.net/forum?id=hZ5XPW7jaD.

[23] Daniel Wang and Andrew Xu. Allocbench: Measuring online tool allocation capability in llm agents, 2026. URL https://arxiv.org/abs/2607.23332.

[24] Jiayu Liu, Cheng Qian, Zhaochen Su, Qing Zong, Shijue Huang, Bingxiang He, and Yi R. Fung. CostBench: Evaluating multi-turn cost-optimal planning and adaptation in dynamic environments for LLM tool-use agents. In Maria Liakata, Viviane P. Moreira, Jiajun Zhang, and David Jurgens, editors, *Proceedings of the 64th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 12826–12858, San Diego, California, United States, July 2026. Association for Computational Linguistics. ISBN 979-8-89176-390-6. doi: 10.18653/v1/2026.acl-long.584. URL https://aclanthology.org/2026.acl-long.584/.

[25] Avanika Narayan, Dan Biderman, Sabri Eyuboglu, Avner May, Scott Linderman, James Zou, and Christopher Re. Cost-efficient collaboration between on-device and cloud language models. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, pages 45682–45719. PMLR, 2025. URL https://proceedings.mlr.press/v267/narayan25a.html.

[26] Siyan Li, Vethavikashini Chithrra Raghuram, Omar Khattab, Julia Hirschberg, and Zhou Yu. PAPILLON: Privacy preservation from Internet-based and local language model ensembles. In *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3371–3390, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.173. URL https://aclanthology.org/2025.naacl-long.173/.

[27] Nenad Tomasev, Matija Franklin, Joel Z. Leibo, Julian Jacobs, William A. Cunningham, Iason Gabriel, and Simon Osindero. Virtual agent economies, 2025. URL https://arxiv.org/abs/2509.10147.

[28] Gagan Bansal, Wenyue Hua, Zezhou Huang, Adam Fourney, Amanda Swearngin, Will Epperson, Tyler Payne, Jake M. Hofman, Brendan Lucier, Chinmay Singh, Markus Mobius, Akshay Nambi, Archana Yadav, Kevin Gao, David M. Rothschild, Aleksandrs Slivkins, Daniel G. Goldstein, Hussein Mozannar, Nicole Immorlica, Maya Murad, Matthew Vogel, Subbarao Kambhampati, Eric Horvitz, and Saleema Amershi. Magentic Marketplace: An open-source environment for studying agentic markets, 2025. URL https://arxiv.org/abs/2510.25779.

[29] Google. Agent Payments Protocol (AP2) specification, v0.2, 2026. URL https://ap2-protocol.org/ap2/specification/. Accessed 2026-11-01.

[30] Google Agentic Commerce. A2A protocol: x402 payments extension v0.2, 2025. URL https://github.com/google-agentic-commerce/a2a-x402/blob/main/spec/v0.2/spec.md. Accessed 2026-11-01.

[31] Model Context Protocol, a Series of LF Projects, LLC. Model Context Protocol specification, version 2026-07-28, 2026. URL https://modelcontextprotocol.io/specification/2026-07-28. Accessed 2026-11-01.

[32] Agent2Agent (A2A) Project. Agent2Agent (A2A) protocol specification, version 1.0.0, 2026. URL https://a2a-protocol.org/v1.0.0/specification/. Accessed 2026-11-01.

[33] Berri AI. LiteLLM, 2026. URL https://github.com/BerriAI/litellm. Software, version 1.106.6. Accessed 2026-11-01.

[34] Ollama. Ollama, 2026. URL https://github.com/ollama/ollama. Software, version 0.40.13. Accessed 2026-11-01.

[35] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 611–626, 2023. doi: 10.1145/3600006.3613165.

[36] LangChain. LangGraph, 2026. URL https://github.com/langchain-ai/langgraph. Software, version 1.2.19. Accessed 2026-11-01.

[37] Presidio Contributors. Presidio: Data protection and de-identification SDK, 2026. URL https://github.com/data-privacy-stack/presidio. Software, version 2.2.366. Originally developed by Microsoft; now maintained by the Data Privacy Stack community. Accessed 2026-11-01.

[38] Yinsicheng Jiang, Liang Cheng, Yeqi Huang, Yufan Zhao, Zhan Lu, Li Dong, Wenda Li, Edoardo Ponti, and Luo Mai. Specialize roles, mix deployments: Pushing the cost-accuracy frontier of LLM agent teams, 2026. URL https://arxiv.org/abs/2606.20629.

[39] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=nZeVKeeFYf9.