ZEBRA: Zero-shot Budgeted Resource Allocation for LLM Orchestration

May Hamri [1] **Inbal Talgam-Cohen** [1]

Abstract

As autonomous agents increasingly execute end-to-end tasks under fixed monetary budgets, the pressing open question shifts from whether the budget is respected, to how to spend it effectively. Existing budget-aware methods typically control reasoning step-by-step within a single agent, or learn resource allocation policies via RL. None address how to split a budget across the composing phases of a multi-agent pipeline at inference time. We propose **ZEBRA**, a zero-shot framework that reduces multi-phase budget allocation to a continuous nonlinear knapsack problem: an LLM controller estimates per-phase utility curves, and a water-filling search on the Lagrange multiplier returns the per-phase split. Additive and multiplicative aggregations are unified under the same solver. On a 150-task APPS coding benchmark, both ZEBRA variants outperform LLM-direct (budget allocation directly by an LLM) on every aggregate metric. At a budget of $\alpha = 0.5$ of the unconstrained spend, ZEBRA recovers 94.4% of unconstrained quality, versus 88.1% for LLM-direct. The advantage is statistically significant and transfers beyond coding: on a 3-phase HotpotQA pipeline, ZEBRA beats LLM-direct by 14.3pp, with allocations empirically robust to curve-estimation noise. On HotpotQA, ZEBRA arrives at a different budget split (near-balanced) compared to the APPS one (skewed towards a refinement phase), showing adaptation to the pipeline structure. More broadly, we show that lightweight algorithmic guidance at inference time can improve the economic behavior of autonomous multi-agent systems.

1. Introduction

As agents increasingly execute end-to-end tasks, resource allocation becomes a first-class requirement for autonomy. Setting limits on resource consumption is crucial – as recently documented, uncontrolled waste of resources emerges in autonomous agentic behavior (Shapira et al., 2026). But users want agents not only to stay within limits on money, tokens, latency, or compute; they also need these resources to be used effectively. E.g., for a given task, the optimal planning depth, tool usage, and verification effort differ considerably under a $10 budget compared with a $100 budget.

The need for effective budget usage is growing in urgency as agent systems become more expensive and complex. Orchestrators like LangGraph or Gas Town (Yegge & contributors, 2026), which coordinate multiple coding agents, illustrate why economic decision-making is becoming a core system capability: the orchestrator must allocate resources judiciously across workers, phases, and tools. Platforms such as Replit or Base44 (which autonomously create apps from prompts) enforce usage limits and billing controls in a brute-force manner (e.g. Base44, 2026). They do not currently provide a controller that optimizes the expected outcome quality under a fixed budget. Some recent works have begun to frame budgeting as an optimization problem, but many of these encode budget reasoning through training-time dynamics (RL controllers, knapsack exploration) rather than inference-time allocation.

We take a complementary route. **ZEBRA** (**ZE**ro-shot **B**udgeted **R**esource **A**llocation) adds an allocation agent to a multi-agent pipeline that, instead of directly outputting per-phase budgets, estimates phase utility curves (concretely, saturating-exponential functions). This results in a continuous nonlinear knapsack problem, which the system solves via bisection on the dual Lagrange multiplier (water-filling). Taken together, we get a zero-shot, inference-time allocation mechanism with no fine-tuning or RL, allowing budgeted

[1]Tel Aviv University, Israel. Correspondence to: May Hamri <mayhamri8@gmail.com>, Inbal Talgam-Cohen <inbal-talgam@gmail.com>.

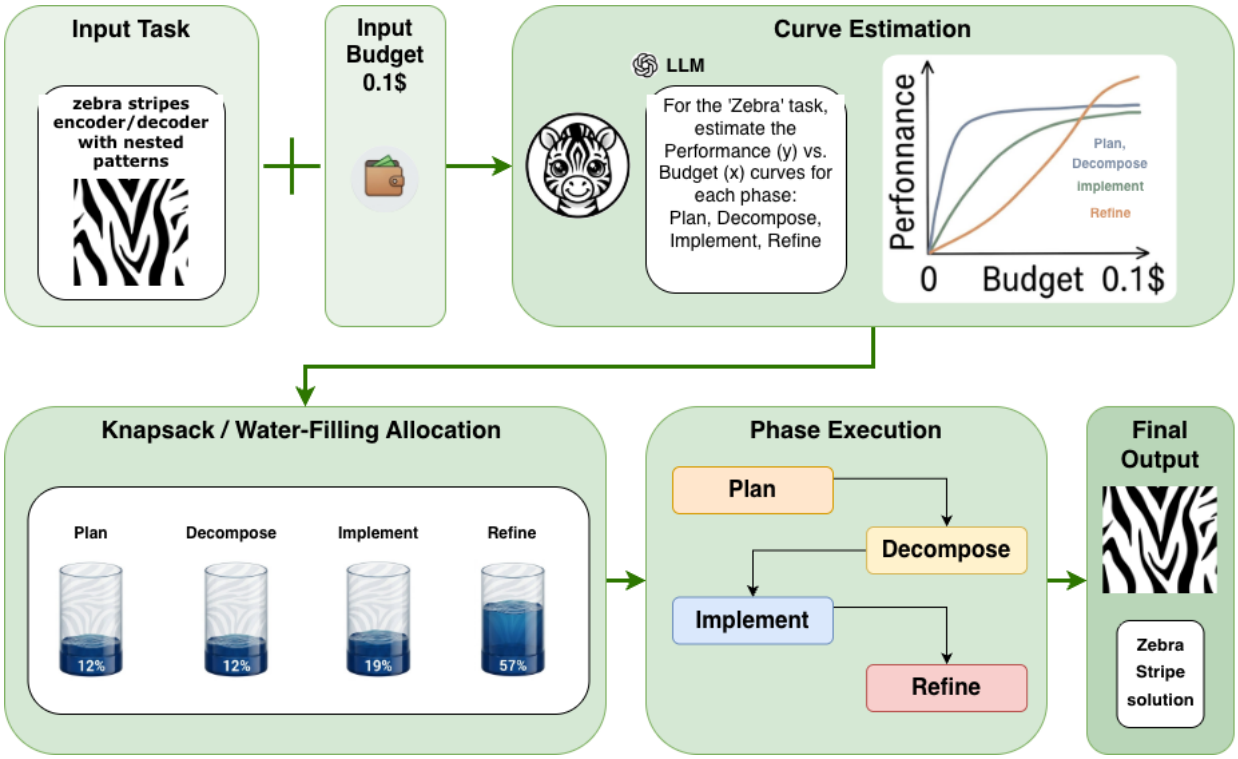

Figure 1. **ZEBRA overview.** Given an input task and a fixed total budget (e.g., $0.10), ZEBRA adds an allocation agent that prompts an LLM to estimate a performance–budget curve for each workflow phase (Plan, Decompose, Implement, Refine). ZEBRA solves a continuous nonlinear knapsack via water-filling to allocate the global budget across phases before execution. The shared multi-phase pipeline then runs under these per-phase budget caps.

execution to approach unconstrained quality at much lower spend. We study two quality aggregation forms over the phases – an additive function and a multiplicative one (with offset curves that still enable phase starvation). Despite their fundamental difference in how overall quality depends on per-phase quality, we unify the multiplicative objective with the additive one under the same solver via a logarithmic transformation.

We summarize our primary contributions: *(i) Method.* We develop a zero-shot, inference-time controller, which allocates a fixed budget across pipeline phases, the quality of which aggregates additively or multiplicatively. This is done by prompting an LLM to estimate phase utility curves, and solving a continuous nonlinear knapsack problem – no fine-tuning or RL required. *(ii) Experiments.* On a 150-task APPS coding interview benchmark (Hendrycks et al., 2021), and across budget regimes $\alpha \in \{0.3, 0.5, 0.8\}$, both ZEBRA objectives outperform direct LLM allocation on every aggregate metric (paired Wilcoxon, BH-adjusted $p < 0.01$), with gains that scale with budget pressure and concentrate on medium and hard tasks (Section 5.2). The same effect reproduces on two additional coding benchmarks: HumanEval (Chen et al., 2021) and CodeContests (Li et al., 2022) at $\alpha = 0.5$ (Section 5.8). Importantly, it also generalizes *beyond coding*: on the 3-phase question-answering HotpotQA pipeline (Yang et al., 2018), ZEBRA beats LLM-direct by $+14.3$pp NB retention ($p = 10^{-4}$) (Section 5.9). ZEBRA also recovers a near-balanced allocation that matches the (here near-optimal) uniform baseline – a distinct allocation shape from the strongly-skewed split it learns on APPS. This is evidence that ZEBRA adapts to, rather than memorizes, the task type. *(iii) Implication.* More broadly, we show that lightweight algorithmic guidance – applied at inference time – can meaningfully improve the economic behavior of autonomous multi-agent systems.

2. Related Work

ZEBRA builds on two distinct bodies of prior work. First, *cost-aware LLM research*, including: (i) the AI provider's cost-saving problem (an inference operator redistributing a shared compute pool across independent user queries), and (ii) multi-agent cost management (selecting, routing, and coordinating agents under a budget). Second, classic *resource allocation theory*, which provides a principled framework for distributing a fixed budget across subprocesses with concave utility curves. ZEBRA combines these by applying the theoretical foundation to a problem that neither line has addressed: allocating a monetary budget by an

orchestrator across the phases of a multi-agent pipeline, the outputs of which jointly compose into a single task outcome, without training data or model modification. Table 3 in Appendix A gives a comparison of related methods along four axes: provider vs. orchestrator budget, discrete vs. continuous spending decisions, zero-shot or not, and dependency of items sharing the budget. Here we overview the main approaches.

Inference-time token budget allocation. Token-budget methods for independent user queries (Han et al., 2025; Brown et al., 2025; Zhao et al., 2026; Lyu et al., 2025) take the *provider* perspective, operate on *independent* queries, and pick *discrete* per-query token budgets; most require task-specific training data or fine-tuning. ZEBRA differs on every axis (Table 3): it is an *orchestration*-side controller that splits a *continuous* monetary budget across the *dependent* phases of a single multi-agent pipeline, zero-shot.

Multi-agent cost management and routing. A parallel line of work reduces cost through discrete model selection or agent routing rather than continuous budget allocation: LLM cascades (Chen et al., 2024), RL-trained per-query model selectors (Jin et al., 2025), ILP-based agent assembly with RL-fixed topology (Yang et al., 2026), VAE-driven workflow synthesis (Su et al., 2026), enterprise pipeline design (Kandogan et al., 2025), and upfront vs. reactive selection (Amayuelas et al., 2025). In every case the decision variable is *discrete and combinatorial* and the controller is trained.

Budget-aware agent control. A concurrent line studies budget-aware control at inference time *inside* a single agent: BATS (Liu et al., 2025) pairs a Budget Tracker plug-in with a per-step "dig deeper vs. pivot" binary decision; INTENT (Liu et al., 2026) trains an intention-aware hierarchical world model for online tool-call risk calibration; and BAVT (Li et al., 2026) scales multi-hop reasoning value-tree nodes by the remaining-resource ratio. All three are *step-by-step* controllers inside a single running agent. ZEBRA targets the complementary, higher-level problem – it determines how much of the total budget is allocated to each agent (rather than how the agent spends its share of the budget).

Resource allocation theory and knapsack methods. Knapsack and its variants provide the theoretical backbone for budget allocation (Katoh & Ibaraki, 1998; Kellerer et al., 2004; Boyd & Vandenberghe, 2004). Boutilier & Lu (2016) use multi-choice knapsack for allocating budget among independent MDPs (predating LLMs); recent LLM uses of knapsack formulations (Zhao et al., 2026; Brown et al., 2025; Li et al., 2025) all operate on *independent* queries or training batches rather than the composing phases of a single running pipeline. In the context of question answering, SEER (Tonglet et al., 2023) packages an LLM system with a discrete knapsack solver for selection of exemplars (pairs of training instances and answers, provided as part of the prompt to assist the LLM in answering questions correctly). The purpose of knapsack here is to control the total prompt size, and the aim is to choose exemplars that are simultaneously diverse and similar to the prompt. In comparison, ZEBRA applies a continuous knapsack solver, and uses it for budget allocation across a pipeline with the aim of maximizing aggregate quality, after informing the solver with learned performance curves. Delegating the allocation step to an external knapsack solver (rather than an LLM) is further motivated by Duchnowski & Pavlick (2025), who document systematic LLM failure on knapsack problems.

3. Problem Formulation

Autonomous agent systems increasingly execute end-to-end tasks under explicit resource constraints (e.g., monetary cost, API credits, token usage, latency, GPU time, or memory). A central challenge is planning actions so that the achieved task quality is maximized subject to a fixed budget, motivating an explicit optimization formulation rather than a behavior learned implicitly by fine-tuning or RL.

3.1. Continuous resource allocation as separable utility maximization

We model an agent or multi-agent pipeline as executing n phases, indexed by $i \in \{1, \ldots, n\}$. Let $B > 0$ denote the total available budget and $x_i \geq 0$ the budget allocated to phase i. Each phase has an associated performance–budget curve $f_i : \mathbb{R}_{\geq 0} \to \mathbb{R}$, where $f_i(x_i)$ quantifies the expected contribution to overall task outcome from spending x_i on phase i. The resulting optimization problem:

$$\max_{\mathbf{x}} \sum_{i=1}^{n} f_i(x_i) \quad \text{s.t.} \quad \sum_{i=1}^{n} x_i \leq B, \;\; x_i \geq 0 \; \forall i. \tag{1}$$

Problem (1) is a classic *single-resource allocation* problem with *separable utilities*, equivalent to the continuous nonlinear knapsack family with "water-filling"-type allocations under appropriate structure assumptions (Katoh & Ibaraki, 1998; Ibaraki & Katoh, 1988; Boyd & Vandenberghe, 2004). We assume each f_i is non-decreasing, concave, and differentiable, so (1) is a convex program with optimality fully characterized by KKT conditions (Boyd & Vandenberghe, 2004) (full assumptions, optional per-phase upper bounds u_i, and the non-convex case are deferred to Appendix B).

3.2. KKT conditions and dual-search algorithm

Introducing a Lagrange multiplier $\lambda \geq 0$ for the budget constraint yields the well-known *marginal equalization* rule $f_i'(x_i^\star) = \lambda$ at every active phase, with the natural bound-

ary modifications when $x_i^\star$ hits 0 or u_i. Economically, λ is the shadow price of budget and matches the water-filling solution from communication systems (Boyd & Vandenberghe, 2004). The total allocation $S(\lambda) = \sum_i x_i(\lambda)$ is non-increasing in λ, so $\lambda^\star$ can be found by bisection in $O(n \log \frac{1}{\varepsilon})$ time, matching the classical resource-allocation result (Katoh & Ibaraki, 1998; Ibaraki & Katoh, 1988) (full derivation in Appendix B). ZEBRA also handles multiplicative objectives via a log transformation that reduces a weighted product $\prod_i h_i(x_i)^{w_i}$ to $\sum_i w_i \log h_i(x_i)$, solvable by the same dual-search (Appendix C).

4. Method

Figure 1 shows how ZEBRA augments a multi-agent pipeline of n phases with an *allocation agent* (controller). Before execution, it assigns each phase a budget $x_i \geq 0$ with $\sum_{i=1}^n x_i \leq B$. Our method is inference-time *algorithmic guidance*: (1) estimate curves; (2) solve for allocations; (3) execute phases under fixed budgets. Figure 2 schematically shows the curves and water-filling solution.

4.1. Parametric phase utility curves

Let $f_i(x)$ denote the expected utility of phase i when spending budget x. To preserve diminishing returns and enable fast optimization, we fit each phase with a concave, saturating exponential:

$$f_i(x) = a_i \left(1 - e^{-b_i x}\right), \tag{2}$$

where $a_i \in (0, 1]$ is the quality ceiling and $b_i > 0$ controls how quickly the phase saturates. This form is monotone and concave for $a_i, b_i > 0$ (derivatives in Appendix B).

Curve-parameter estimation via two-point fitting. Rather than asking the controller for abstract curve parameters, we elicit two intuitive operating points per phase: $n_{\text{basic},i}$, the total output tokens this phase needs to reach roughly 50% of its potential quality, and $n_{\text{great},i}$, the total output tokens to reach roughly 90% (named `tokens_basic` and `tokens_great` in the controller prompt; see Appendix G.4). We fit b_i as the geometric mean of the two implied rates and ask the controller to estimate the ceiling $a_i \in (0, 1]$ alongside b_i (full equations and the propagation-weighted variant in Appendices B and D.12). Token operating points are converted to USD via per-phase model pricing before b_i is computed, so b_i has units of inverse-USD and matches the monetary budget B.

4.2. Allocation objectives

Given the estimated curves $\{(a_i, b_i)\}_{i=1}^n$, we consider two ways to aggregate per-phase utilities into a pipeline-level objective: an independent sum and a multiplicative product. Both reduce to a separable concave program with a single budget constraint, so both are solved by the same Lagrangian dual-search of Section 3.2: bisect on a single shadow price λ until the per-phase responses sum to B. The two objectives differ only in the closed form of $x_i(\lambda)$.

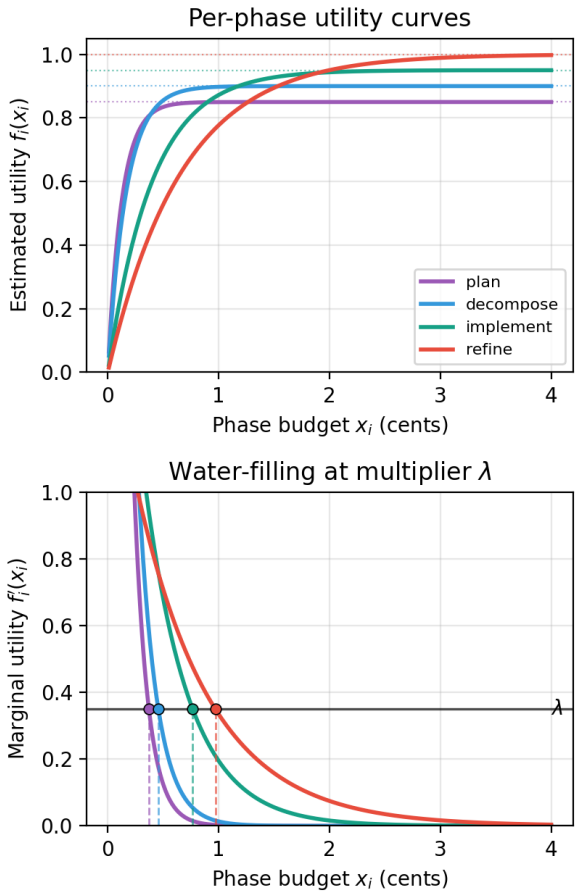

Figure 2. **Per-phase utility curves and water-filling.** *Left:* the saturating exponential $f_i(x) = a_i(1 - e^{-b_i x})$ for the four pipeline phases, with the quality ceiling a_i shown as a dotted line. Phases differ in how quickly they saturate (the b_i parameter): `plan` reaches its ceiling fastest, `refine` slowest. *Right:* the corresponding marginal-utility curves $f_i'(x) = a_i b_i e^{-b_i x}$. ZEBRA's knapsack solution is precisely water-filling: pick a single Lagrange multiplier λ (horizontal line), and read off each phase's budget x_i where its marginal-utility curve crosses λ. The total budget constraint $\sum_i x_i \leq B$ is enforced by binary search on λ.

4.2.1. Additive objective

The additive formulation assumes phase contributions are independent and combine as a sum:

$$\max_{\mathbf{x}} \sum_{i=1}^n a_i\left(1 - e^{-b_i x_i}\right) \quad \text{s.t.} \quad \sum_{i=1}^n x_i \leq B, \;\; x_i \geq 0 \; \forall i. \tag{3}$$

Setting $f_i'(x_i) = \lambda$ at every active phase and solving gives the closed-form per-phase response $x_i(\lambda) =$

$\max\big(0,\ \frac{1}{b_i}\ln\frac{a_i b_i}{\lambda}\big)$, with λ as the shadow price of the budget. Phase i is starved exactly when its *marginal at zero* $f_i'(0) = a_i b_i$ falls below this price – i.e., the first dollar spent on phase i buys less pipeline quality than the same dollar spent elsewhere.

4.2.2. Multiplicative offset objective

A multiplicative aggregator captures the intuition that the pipeline behaves as a chain: a failure in any phase degrades the whole task. The naïve choice $\prod_i f_i(x_i)$ has a problematic corner $f_i(0) = 0$, so any unfunded phase forces the product to zero, and the optimizer is forced to fund every phase. We restore the ability to starve phases by replacing f_i with the *offset* curve

$$g_i(x) \;=\; (1-a_i) \;+\; a_i\big(1-e^{-b_i x}\big) \;=\; 1-a_i\, e^{-b_i x}, \quad (4)$$

i.e., $f_i(x)$ shifted up by a baseline $1 - a_i$. Concretely, g_i ranges from $g_i(0) = 1 - a_i$ to $g_i(\infty) = 1$, so it can be read as a per-phase *pass-through quality*: a phase that gets no budget still contributes the baseline $1 - a_i$, and approaches a perfect 1 as more budget is spent. The pipeline-level objective is

$$\max_{\mathbf{x}} \prod_{i=1}^{n} g_i(x_i) \;=\; \prod_{i=1}^{n}\big(1 - a_i\, e^{-b_i x_i}\big)$$
$$\text{s.t.} \quad \sum_i x_i \le B, \;\; x_i \ge 0, \quad (5)$$

i.e., the product of these pass-through qualities, which lies in $[\prod_i(1 - a_i),\, 1]$. Phases with high ceilings (large a_i) hurt the product most when starved and therefore attract budget; phases with a_i near zero already have $g_i(0)$ near 1 and can be skipped at almost no cost. Taking logs (see Appendix C) reduces (5) to the separable concave program $\max \sum_i \log g_i(x_i)$, solvable by the same dual-search; the closed-form per-phase response and starvation threshold are derived in Appendix B.

Sum vs. product: different starvation criteria. The two objectives differ in *which* phases they starve when the budget binds. The additive objective starves phases with low marginal at zero (small $a_i b_i$), due to "low return on the first dollar." The multiplicative offset objective starves phases with low ceiling (small a_i), since "baseline pass-through is already near 1, so the product is fine without budget." These two policies thus behave differently when the budget binds (Section 5.2). A propagation-weighted extension is in Appendix D.12.

5. Experiments

5.1. Setup

Question. Given a fixed multi-phase coding workflow and total budget, does estimating phase utility curves and solving for allocations beat asking an LLM to split the budget directly? We hold the pipeline, models, and budget constant and vary only the `allocate` component.

Benchmark. We evaluate on the APPS benchmark (Hendrycks et al., 2021) at the *interview* difficulty level. We construct a benchmark of 150 tasks balanced across difficulty tiers (50 each), where difficulty is defined by the unconstrained pipeline's mean score: *easy* ($\bar{s} \ge 0.8$), *medium* ($0.4 \le \bar{s} < 0.8$), *hard* ($\bar{s} < 0.4$). Tasks are screened for cost stability ($\mathrm{CV}_{\text{cost}} < 0.35$ over 30 unconstrained runs) so that per-task budget references are meaningful. We additionally run two transfer benchmarks at $\alpha = 0.5$ – HumanEval (Chen et al., 2021) and CodeContests (Li et al., 2022) – with the same protocol (Section 5.8). Construction details are in Appendix D.

Budget levels. For each task we set the total budget relative to its mean unconstrained cost $\bar{c}$, with $B = \alpha\,\bar{c}$. We evaluate every strategy at $\alpha \in \{0.5, 0.8\}$ – the *tight* and *moderate* regimes – and additionally run a *very-tight* regime $\alpha = 0.3$ on the 50 easy tasks.

Strategies. We compare the two ZEBRA objectives – `additive` (Sec. 4.2.1) and `mult_offset` (Sec. 4.2.2) – against **LLM-direct**: the controller, given the same pipeline description and total budget, outputs a per-phase split. This baseline asks whether an LLM can self-allocate budget via internal reasoning instead of an explicit solver – a non-trivial question given documented LLM weakness on NP-hard combinatorial problems including knapsack (Duchnowski & Pavlick, 2025), which had not been measured in the orchestration setting (where the model can introspect over its own pipeline structure). All controller-based strategies share the controller model and prompt scaffold and differ only in how the allocation is produced. **no_budget**, the unconstrained pipeline, is the oracle ceiling.

Pipeline and models. The workflow is a LangGraph `StateGraph` (LangChain, 2026a;b): `allocate` → `plan` → `decompose` → `implement` → `refine` → `END`. The graph is identical across strategies; only `allocate` differs. Base phases use `gpt-4o-mini`; `refine` (a review→revise loop, up to 3 iterations) and the controller use `gpt-4o`. Budget enforcement uses a budget-capped LLM interface, and the pipeline keeps the best candidate across iterations so additional iterations never degrade output. See Appendix D.1 and D.5.

Evaluation. For each (task, α, strategy) cell we perform 15 independent runs and report mean and median per-task score, task-averaged success rate (score > 0.5), realized cost, and *NB retention* (mean score divided by the no-budget mean on the same tasks). Significance vs. LLM-direct uses paired Wilcoxon signed-rank tests on per-task mean scores. Across the 18 ZEBRA-vs-LLM-direct comparisons of the

main APPS family (App. D.4, Table 5) we control FDR at $q = 0.05$ via Benjamini–Hochberg (Benjamini & Hochberg, 1995); significance markers reflect BH-adjusted p-values.

5.2. Main results

Table 1 reports the overall comparison on all 150 tasks. *Both ZEBRA objectives outperform LLM-direct at both budget levels on every aggregate metric we track.* At the tight budget ($\alpha = 0.5$), `mult_offset` recovers 94.4% of the unconstrained score, versus 88.1% for LLM-direct – a 6.3-point retention gap at essentially identical spend ($0.011 per task). At the moderate budget ($\alpha = 0.8$), `additive` recovers 98.1% versus 94.3% for LLM-direct. Both ZEBRA variants are significant against LLM-direct at $p < 0.01$ (paired Wilcoxon, $n = 150$); the strongest variant at each budget level achieves $p < 0.001$.

5.3. The advantage concentrates on medium and hard tasks

Table 2 breaks the same results down by difficulty tier. **Easy tasks tie** (all strategies recover $\approx$ 93–96% of unconstrained quality; no ZEBRA-vs-LLM difference is significant); **the gap opens on medium and hard tasks**: on medium tasks at $\alpha = 0.5$ the best ZEBRA variant lifts mean score by +6.7 points (+10.0 pp success rate) over LLM-direct ($p_{\text{BH}} = 3.7 \times 10^{-4}$); on hard tasks ZEBRA wins by +3.4 points ($p_{\text{BH}} < 0.01$). The same pattern holds at $\alpha = 0.8$, with significance preserved on every medium/hard cell. Allocation quality matters only when the budget binds; a per-task scatter view (Fig. 4, App. D.8) confirms easy tasks cluster on the diagonal while harder tasks drift above it.

5.4. Budget pressure: ZEBRA's gap scales with tightness

To test whether the easy-tier tie is about the tasks being easy or the budget being generous, we rerun the 50 easy tasks at a very-tight budget $\alpha = 0.3$. The picture flips: `zebra-additive` now beats LLM-direct on per-task mean ($p_{\text{BH}} = 0.030$; full numbers in App. D.9). Combined with the tier breakdown this gives a simple rule of thumb: *ZEBRA's advantage over LLM allocation scales with budget tightness* – whether tightness comes from a harder task or a smaller α (Fig. 5, App. D.9).

5.5. Why ZEBRA wins: it adapts the spend

ZEBRA spends *differently*: Figure 6 and Table 9 (Appendix D.10) show the average fraction of the budget spent on each phase by tier. **ZEBRA adapts to task difficulty; LLM-direct does not.** On easy tasks at $\alpha = 0.5$, ZEBRA assigns `refine` only 30–45% of the budget; on medium and hard tasks, both variants pull strongly toward `refine` (60–66%). LLM-direct, by contrast, spends an essentially fixed ∼25% on `implement` and ∼50% on `refine` regardless of tier or budget. At $\alpha = 0.3$ on easy tasks, ZEBRA variants compress `refine` still further (10.7% for `additive`, 19.3% for `mult_offset`), while LLM-direct keeps it near 47% (Table 8, Appendix D.9). This adaptation is the likely cause of the score gaps in Table 2 and the retention scaling in Figure 5.

Additive vs. mult_offset. `additive` is the per-task win-count leader at every aggregate budget (64/150 at $\alpha = 0.5$; Table 10). `mult_offset` wins where its product form matters: on hard tasks at $\alpha = 0.5$ it reaches 0.130 mean (vs. 0.123 `additive`, 0.096 LLM-direct) and takes 21/50 wins. `mult_offset`'s product form drags the objective down whenever any single phase is under-funded, so the optimizer concentrates budget on the phase that would otherwise be the weakest link. This bias only pays off when the budget is too tight to fund every phase well *and* the task can't tolerate a weak phase, which is precisely the hard-tier, $\alpha = 0.5$ cell.

5.6. Ablations

We run four ablations against `zebra-mult_offset` on the same 150 APPS tasks (full numbers in App. D.6). **(1) ZEBRA-LLM** – swap the knapsack solver for the controller given the *same* fitted curves: −4.20 to −4.31 points across budgets. **(2) Uniform** (25/25/25/25) – drop both curves and solver: −5.14 to −5.92 points (both $p < 10^{-4}$, $n = 150$). **(3) LLM-CoT** rules out a prompting confound: a CoT scaffold on LLM-direct at $\alpha = 0.5$ scores 0.0070 worse than plain LLM-direct ($p_{\text{raw}} = 0.054$, n.s.), and ZEBRA still beats CoT by +0.0429 ($p_{\text{raw}} < 10^{-4}$). **(4) Fixed-Avg** bakes in ZEBRA's global mean per-phase share (11.3/14.0/24.1/50.6%) and applies it to every task, isolating per-task adaptation: it matches dynamic ZEBRA on easy tasks but loses +9 to +22 pp NB-retention on medium/hard, with the gap statistically significant in 3 of the 4 medium/hard cells ($p < 0.01$); the exception is `additive` vs. Fixed-Avg on hard, $p = 0.17$ (App. D.16). All four confirm that curves, solver, allocator, *and* per-task adaptation are each required.

5.7. Robustness, overhead, and assumptions

Methodology and full details are in Appendices D.14-D.15, D.17-D.18. **(i) Curve-estimation noise.** Multiplicative Gaussian noise on (a_i, b_i) at $\sigma \in \{0, 10, 50\}\%$ leaves performance close to the live controller. At $\sigma = 50\%$ (parameters halved/doubled), the mean score drops only −1.4pp ($p = 0.08$ vs. live, n.s.) and still beats Uniform ($p = 0.03$). Swapping the controller LLM (`gpt-4o` ↔ `gpt-4o-mini`) leaves per-task fractional allocations

α	Strategy	Mean score	Median	Success rate	Cost ($)	NB retention
0.5	zebra-additive	0.5258	0.5333	52.7%	0.0134	92.0%**
	zebra-mult_offset	**0.5397**	**0.5362**	**54.1%**	0.0110	**94.4%***
	llm	0.5038	0.5026	49.7%	0.0109	88.1%
	no_budget (oracle)	0.5717	—	—	—	100.0%
0.8	zebra-additive	**0.5605**	**0.5972**	**56.2%**	0.0192	**98.1%***
	zebra-mult_offset	0.5576	0.5957	55.2%	0.0167	97.5%**
	llm	0.5389	0.5323	54.0%	0.0140	94.3%
	no_budget (oracle)	0.5717	—	—	—	100.0%

Table 1. **Main results on** 150 **APPS interview-level tasks** (15 runs per cell). Mean and median are taken over per-task mean scores; success rate is the task-averaged fraction of runs with score > 0.5; NB retention is computed as a ratio of means, $\bar{s}/\bar{s}_{\text{NB}}$ (the strategy's mean score on the task set divided by the no-budget mean score on the same set). The two ZEBRA variants differ in the allocation objective (sum vs. product). Significance markers on NB retention come from a paired Wilcoxon signed-rank test on per-task mean score against LLM-direct, with p-values BH-adjusted within the 18-test APPS main family (Appendix D.4, Table 5; $^{*}p_{\text{BH}} < 0.05$, $^{**}p_{\text{BH}} < 0.01$, $^{***}p_{\text{BH}} < 0.001$). Bold marks the best ZEBRA variant per column within each α block. **Both ZEBRA variants outperform LLM-direct at both budget levels on every aggregate metric.**

Tier	Strategy	$\alpha = 0.5$			$\alpha = 0.8$		
		Mean	Succ%	p vs LLM	Mean	Succ%	p vs LLM
Easy ($n = 50$)	zebra-additive	0.9216	94.3%	0.19 ns	0.9556	**98.1%**	0.61 ns
	zebra-mult_offset	**0.9385**	**95.9%**	0.18 ns	**0.9582**	97.9%	0.27 ns
	llm	0.9325	94.8%	—	0.9505	97.5%	—
Medium ($n = 50$)	zebra-additive	0.5324	59.6%	0.0025**	**0.5847**	**65.5%**	0.0043**
	zebra-mult_offset	**0.5506**	**62.7%**	0.0000***	0.5772	64.3%	0.022*
	llm	0.4833	52.7%	—	0.5428	61.1%	—
Hard ($n = 50$)	zebra-additive	0.1234	**4.3%**	0.0042**	**0.1412**	**5.1%**	0.0041**
	zebra-mult_offset	**0.1300**	3.9%	0.0044**	0.1376	3.6%	0.019*
	llm	0.0956	1.7%	—	0.1233	3.6%	—

Table 2. **Results by difficulty tier.** Mean score is the per-task mean score averaged over the tier; success rate is the fraction of runs with score > 0.5, task-averaged. The p **vs LLM** column reports the two-sided paired Wilcoxon signed-rank p-value on per-task mean score against LLM-direct ($n = 50$ per tier; $^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$, ns = not significant). Differences are non-significant on easy tasks (both methods saturate within ± 0.6 points of each other) and become significant for both ZEBRA variants on the medium and hard tiers. The largest jumps appear on *medium* tasks at $\alpha = 0.5$, where `mult_offset` lifts mean score by $+6.7$ points ($+10.0$ pp success) over LLM-direct. Bold marks the best strategy per column within each tier $\times$ α cell.

highly correlated (per-phase-averaged Pearson 0.95 on `additive`, 0.92 on `mult_offset`; App. D.15). A post-hoc calibration check of the controller's curve estimates against the per-call quality gains actually observed in run logs shows the predictions track the measured curve shape well (App. D.7, Fig. 3). **(ii) Controller overhead.** Curve estimation costs $\sim$33% of the budget at $\alpha = 0.5$ for both `additive` (29.1% of total spend) and `mult_offset` (29.2%); full per-strategy table in App. D.17. To check that the allocation gain justifies this overhead we compare ZEBRA at $\alpha = 0.5$ against *Uniform at* $\alpha = 0.8$ (60% more budget, zero overhead); both ZEBRA variants still win significantly ($+0.027$, $p = 1.6{\times}10^{-4}$ for `additive`; $+0.041$, $p < 10^{-6}$ for `mult_offset`). **(iii) Pipeline monotonicity.** Recent work on overthinking (Chen et al., 2026; Sui et al., 2025) shows non-decreasing utility can fail for individual prompts. However, within-task fixed-effects regressions on 18,000 runs yield positive per-tier slopes ($\hat{\beta}_{\text{easy}} = 9.4$, $\hat{\beta}_{\text{med}} = 5.2$, $\hat{\beta}_{\text{hard}} = 1.8$; all $p < 10^{-6}$), validating the assumption for our pipeline.

5.8. Transfer to HumanEval (Chen et al., 2021) and CodeContests (Li et al., 2022)

We run two additional benchmarks at $\alpha = 0.5$ (Appendix D.13): HumanEval (29 stable tasks; 28E), and CodeContests (23 tasks; 8E/5M/10H). On HumanEval, `zebra-mult_offset` beats LLM-direct by $+0.025$ ($p_{\text{raw}} = 0.025$, W/L/T = 11/1/17); on CodeContests, both ZEBRA variants finish above LLM-direct in mean and NB-retention, but neither reaches significance at $n = 23$. However, the overall signal is consistent with APPS: gains concentrate where the budget is most binding.

5.9. Generality: a non-coding domain and a different pipeline (HotpotQA)

The HumanEval and CodeContests transfers above hold the *coding* domain and a 4-phase LangGraph pipeline fixed; they test whether ZEBRA generalizes across coding benchmarks but not whether it generalizes beyond coding or the orchestration framework we built it on. To stress this we run a third transfer that changes both: multi-hop question answering on HotpotQA's (Yang et al., 2018), with a plain-Python (no LangGraph) pipeline of three phases: `direct` (one-shot answer) → `research` (sub-question decomposition that yields *evidence only*, not a candidate answer) → `synthesize` (combine the direct answer with the evidence, keeping the direct answer unless contradicted). We screen 200 multi-hop tasks with the same protocol and retain 48 stable tasks (Appendix E); we compare both ZEBRA objectives, LLM-direct, and a Uniform $\frac{1}{3}/\frac{1}{3}/\frac{1}{3}$ baseline at $\alpha = 0.5$, 15 runs per cell, with token-F1 against the gold answer as the score. The solver and controller code are unchanged from APPS. Full numbers are in Table 20; per-phase allocations are in Table 21.

ZEBRA significantly beats LLM-direct on a non-coding pipeline. `zebra-additive` retains 92.1% of the no-budget F1 versus 77.8% for LLM-direct, a +14.3pp NB-retention gap (corresponding to +0.073 on absolute mean F1: 0.473 vs. 0.400; $p = 1 \times 10^{-4}$, paired Wilcoxon on per-task mean F1, $n = 48$); the gap widens to +19.6pp NB retention on the hard tier ($p = 0.0016$). LLM-direct again uses a fixed, ill-calibrated split – this time overweighting `direct` ($\sim 68\%$) and starving `synthesize` ($\sim 12\%$) – so the same failure mode we see on APPS reproduces on a domain and pipeline structure with no shared phases or framework code (Table 21).

Interesting finding: ZEBRA and Uniform are statistically indistinguishable. `zebra-additive` vs. Uniform: +0.8pp NB retention overall (+0.004 on absolute mean F1; $p = 0.65$); per tier −1.7pp easy ($p = 0.86$) and +3.4pp hard ($p = 0.44$). Unlike APPS, where the right allocation is strongly skewed toward `refine` and Uniform leaves ~ 5 points on the table, on HotpotQA the optimal allocation is close to balanced – the three QA phases contribute comparably to the final F1, and any near-balanced split lands close to the right answer. Crucially, *ZEBRA discovers this on its own*: averaged across all 48 tasks its allocation is 30/39/30 (Table 21), within a few points of the 33/33/33 uniform reference. On APPS the same controller had learned a sharply skewed $\sim$ 11/14/24/51 split; here it converges back toward balance because that is what the task structure asks for.

Allocation is also consistent *across tiers*, unlike on APPS. On APPS, ZEBRA's per-phase shares move visibly between easy and medium/hard tasks – `refine` grows from $\sim$ 30–45% on easy to $\sim$ 60–66% on medium/hard (Sec. 5.5, Table 9). On HotpotQA the allocation is essentially *flat* across tiers: the per-phase shares change by less than $\sim$ 2pp from easy to hard (Table 21; `additive` 31/39/30 easy vs. 29/40/31 hard). Non-adaptation here is correct: on this benchmark the optimal split does not change with difficulty, and ZEBRA reflects that. The framework adapts when this is warranted (APPS), and stays balanced otherwise (HotpotQA), without that behavior being hard-coded.

6. Conclusion and Limitations

On APPS, both ZEBRA objectives outperform direct LLM allocation, and the gap widens as the budget tightens. This reproduces on HumanEval/CodeContests, and on a non-coding HotpotQA pipeline. The result also survives 50% curve noise, controller-LLM swaps, and a Uniform comparator given 60% more budget. Studying the allocations shows *why*: ZEBRA adapts its phase split to task type, task difficulty, and budget pressure – skewed toward `refine` on APPS, near-balanced on HotpotQA – while LLM-direct's split is fixed and miscalibrated in both regimes. More broadly, a small amount of optimization-theoretic logic applied at inference time can noticeably improve the economic behavior of autonomous multi-agent systems.

Limitations. ZEBRA allocates once before execution; recursive/online reallocation and budget-aware downstream agents (rather than capped LLM interfaces) are natural extensions. We did explore a hybrid mid-pipeline re-allocation variant on the APPS pipeline, but it did not improve in our ZEBRA case (Appendix F). The $\sim$33% controller overhead is empirically justified (Appendix D.17), but can strain prohibitive budgets. The framework presupposes a pipeline of discrete phases, limiting applicability to phaseless agentic loops.

Future work. Several extensions are natural but not addressed here: workflows with branching/merging structure expressed as DAGs; per-phase model choice; and explicit prefill/decode cost separation. Replication on open-weight controllers is also left for future work.

Acknowledgments

We thank Rana Shahout for her thoughtful and insightful feedback, and Eden Saig for his helpful comments and suggestions. This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 101077862, project ALGOCONTRACT), by the Israel Science Foundation (grant No. 3331/24), by the NSF-BSF (grant No. 2021680), by a Google Research Scholar Award, by a Microsoft AIEI fellowship, and by the AISI Alignment Project.

LLM/Agent Usage Disclosure

In line with the workshop's policy, we disclose the nature and extent of LLM/agent usage in this work. We distinguish two roles: LLMs as the *object of study* (the system being evaluated) and LLMs as *research assistants* to the authors.

LLMs as the object of study. The pipeline evaluated in this paper is itself an LLM-based multi-agent system. The controller and the four pipeline phases (Plan, Decompose, Implement, Refine) call OpenAI models – specifically `gpt-4o-mini` for the base phases and `gpt-4o` for the controller and the `refine` review-revise loop – through the official OpenAI API. Exact model snapshots, prompts, and the LangGraph `StateGraph` wiring are documented in Section 5.1 and Appendix D.

LLMs/agents as research assistants. We made substantial use of general-purpose LLM and coding-agent tools during the research process. The tools used were primarily **Claude** (Anthropic), with additional use of **ChatGPT** (OpenAI) and **Cursor** as a coding assistant integrated with the authors' editor. They assisted across the following stages:

- **Writing.** Drafting, editing, and proof-reading of the manuscript text, including English-language phrasing, paragraph-level restructuring, and LaTeX formatting (tables, figures, bibliography entries, cross-references).
- **Literature analysis and related work.** Searching for and summarising candidate references, surfacing themes for the related-work sections, and helping group prior work into the thematic clusters used in Section 2. All cited claims and citations were verified by the authors against the original sources.
- **Code.** Writing, refactoring, and debugging the experimental code (pipeline orchestration, knapsack solver, benchmark construction scripts, and analysis scripts that produced the tables and statistical tests reported in the paper).
- **Figure generation.** Generating and iterating on the plots in the paper (e.g. the pressure-retention curve in Figure 5, the per-task scatter, and allocation-share bar charts).

Verification and oversight. All LLM-assisted outputs were reviewed by the authors before being incorporated. Code was executed and the results were inspected; numeric results in tables and figures come from actual pipeline runs rather than from LLM-generated text. Citations and quoted claims were verified against the original sources. The authors take full responsibility for the final content of the paper, the experimental design, the analysis, and the conclusions drawn.

References

Amayuelas, A., Yang, J., Agashe, S., Nagarajan, A., Antoniades, A., Wang, X. E., and Wang, W. Self-resource allocation in multi-agent LLM systems, 2025. URL https://arxiv.org/abs/2504.02051.

Bai, M. and Cardonha, C. Cardinality-constrained continuous knapsack problem with concave piecewise-linear utilities, 2023. URL https://arxiv.org/abs/2302.03781.

Base44. Billing and plans - Base44 support documentation, 2026. URL https://docs.base44.com/Account-and-billing/Billing-and-plans. Accessed: 2026-03-14.

Benjamini, Y. and Hochberg, Y. Controlling the false discovery rate: A practical and powerful approach to multiple testing. *Journal of the Royal Statistical Society. Series B (Methodological)*, 57(1):289–300, 1995. URL https://www.jstor.org/stable/2346101.

Boutilier, C. and Lu, T. Budget allocation using weakly coupled, constrained Markov decision processes. In *Proceedings of the 32nd Conference on Uncertainty in Artificial Intelligence (UAI)*, 2016. URL https://www.cs.toronto.edu/~cebly/Papers/DBMDPs_uai.pdf.

Boyd, S. and Vandenberghe, L. *Convex Optimization*. Cambridge University Press, 2004.

Brown, K., Muppidi, A., and Shahout, R. Predictive scheduling for efficient inference-time reasoning in large language models. In *ES-FoMo III: Workshop on Efficient Systems for Foundation Models (ICML 2025)*, 2025. URL https://arxiv.org/abs/2602.01237.

Chen, L., Zaharia, M., and Zou, J. FrugalGPT: How to use large language models while reducing cost and improving performance. *Transactions on Machine Learning Research*, 2024. URL https://arxiv.org/abs/2305.05176.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al. Evaluating large language models trained on code, 2021. URL https://arxiv.org/abs/2107.03374.

Chen, X., Xu, J., Liang, T., He, Z., Pang, J., Yu, D., Song, L., Liu, Q., Zhou, M., Zhang, Z., Wang, R., Tu, Z., Mi, H., and Yu, D. Do NOT think that much for 2+3=? on the overthinking of o1-like LLMs. In *Proceedings of the 43rd International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2412.21187.

Duchnowski, A. and Pavlick, E. A knapsack by any other name: Presentation impacts LLM performance on NP-hard problems. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, 2025. URL https://arxiv.org/abs/2502.13776.

Han, T., Wang, Z., Fang, C., Zhao, S., Ma, S., and Chen, Z. Token-budget-aware LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL 2025*, 2025. URL https://arxiv.org/abs/2412.18547.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Zou, A., Song, D., and Steinhardt, J. Measuring coding challenge competence with APPS. In *Proceedings of the 35th Conference on Neural Information Processing Systems (NeurIPS)*, 2021. URL https://arxiv.org/abs/2105.09938.

Hua, Y., Chen, H., Wang, S., Li, W., Wang, X., and Luo, J. Shapley-Coop: Credit assignment for emergent cooperation in self-interested LLM agents. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2025. URL https://arxiv.org/abs/2506.07388.

Ibaraki, T. and Katoh, N. *Resource Allocation Problems: Algorithmic Approaches*. MIT Press, 1988.

Jin, B., Collins, T., Yu, D., Cemri, M., Zhang, S., Li, M., Tang, J., Qin, T., Xu, Z., Lu, J., Yin, G., Han, J., and Wang, Z. Controlling performance and budget of a centralized multi-agent LLM system with reinforcement learning, 2025. URL https://arxiv.org/abs/2511.02755.

Kandogan, E., Bhutani, N., Zhang, D., Chen, R. L., Gurajada, S., and Hruschka, E. Orchestrating agents and data for enterprise: A blueprint architecture for compound AI, 2025. URL https://arxiv.org/abs/2504.08148.

Katoh, N. and Ibaraki, T. Resource allocation problems. In *Handbook of Combinatorial Optimization*. Springer, 1998.

Kellerer, H., Pferschy, U., and Pisinger, D. *Knapsack Problems*. Springer, 2004. doi: 10.1007/978-3-540-24777-7.

LangChain. Langgraph graph API (Python) documentation, 2026a. URL https://docs.langchain.com/oss/python/langgraph/graph-api. Accessed: 2026-03-07.

LangChain. Langgraph `StateGraph` API reference (Python), 2026b. URL https://reference.langchain.com/python/langgraph/graph/state/StateGraph. Accessed: 2026-03-07.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Dal Lago, A., et al. Competition-level code generation with AlphaCode. *Science*, 378(6624):1092–1097, 2022. doi: 10.1126/science.abq1158. URL https://www.science.org/doi/10.1126/science.abq1158.

Li, Y., Deng, W., Li, J., and Li, X. Spend less, reason better: Budget-aware value tree search for LLM agents, 2026. URL https://arxiv.org/abs/2603.12634.

Li, Z., Chen, C., Yang, T., Ding, T., Sun, R., Zhang, G., Huang, W., and Luo, Z.-Q. Knapsack RL: Unlocking exploration of LLMs via optimizing budget allocation, 2025. URL https://arxiv.org/abs/2509.25849.

Liu, H., Tian, C., An, N., Wang, Z., Lu, P., Yu, C., and Qi, Q. Budget-constrained agentic large language models: Intention-based planning for costly tool use, 2026. URL https://arxiv.org/abs/2602.11541.

Liu, T., Wang, Z., Miao, J., Hsu, I.-H., Yan, J., Chen, J., Han, R., Xu, F., Chen, Y., Jiang, K., Daruki, S., Liang, Y., Wang, W. Y., Pfister, T., and Lee, C.-Y. Budget-aware tool-use enables effective agent scaling, 2025. URL https://arxiv.org/abs/2511.17006.

Lyu, S., Wu, L., Yan, Y., Wu, X., Li, H., Shen, Y., Jiang, P., Lu, W., Xiao, J., and Zhuang, Y. Hierarchical budget policy optimization for adaptive reasoning, 2025. URL https://arxiv.org/abs/2507.15844.

Rajpurkar, P., Zhang, J., Lopyrev, K., and Liang, P. SQuAD: 100,000+ questions for machine comprehension of text. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2383–2392, 2016. URL https://arxiv.org/abs/1606.05250.

Shapira, N., Wendler, C., Yen, A., Sarti, G., Pal, K., Floody, O., Belfki, A., Loftus, A., Jannali, A. R., Prakash, N., Cui, J., Rogers, G., Brinkmann, J., Rager, C., Zur, A., Ripa, M., Sankaranarayanan, A., Atkinson, D., Gandikota, R., Fiotto-Kaufman, J., Hwang, E., Orgad, H., Sahil, P. S., Taglicht, N., Shabtay, T., Ambus, A., Alon, N., Oron, S., Gordon-Tapiero, A., Kaplan, Y., Shwartz, V., Shaham, T. R., Riedl, C., Mirsky, R., Sap, M., Manheim, D., Ullman, T., and Bau, D. Agents of chaos, 2026. URL https://arxiv.org/abs/2602.20021.

Su, J., Lan, Q., Xia, Y., Sun, L., Tian, W., Shi, T., Song, X., He, L., and Yang, J. Difficulty-aware agentic orchestration for query-specific multi-agent workflows. In *Proceedings of the ACM Web Conference 2026 (WWW '26)*, 2026. URL https://arxiv.org/abs/2509.11079.

Sui, Y., Chuang, Y.-N., Wang, G., Zhang, J., Zhang, T., Yuan, J., Liu, H., Wen, A., Zhong, S., Chen, H., and Hu, X. Stop overthinking: A survey on efficient reasoning for large language models. *Transactions on Machine Learning Research*, 2025. URL https://arxiv.org/abs/2503.16419.

Tonglet, J., Reusens, M., Borchert, P., and Baesens, B. SEER: A knapsack approach to exemplar selection for in-context HybridQA. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 2023. URL https://arxiv.org/abs/2310.06675.

Yang, L., Luo, J., Liu, X., Lou, Y., and Chen, Z. BAMAS: Structuring budget-aware multi-agent systems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 2026. URL https://arxiv.org/abs/2511.21572.

Yang, Z., Qi, P., Zhang, S., Bengio, Y., Cohen, W. W., Salakhutdinov, R., and Manning, C. D. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 2369–2380, 2018. URL https://arxiv.org/abs/1809.09600.

Yegge, S. and contributors. Gas Town - multi-agent workspace manager, 2026. URL https://github.com/steveyegge/gastown. Accessed: 2026-03-14.

Zhao, M., Qi, Q., and Sun, H. ROI-reasoning: Rational optimization for inference via pre-computation metacognition, 2026. URL https://arxiv.org/abs/2601.03822.

A. Extended Related Work

A.1. Per-axis comparison with prior work

Table 3 contrasts ZEBRA with the closest budgeted-LLM and resource-allocation methods discussed in Section 2 along four axes: **Perspective** – whether the budget is set from the *provider* side (an inference operator splitting compute across many independent user queries) or the *orchestration* side (a controller allocating budget within a single multi-agent task); **Decision (params)** – the variable each method picks (its scope and form, including whether it is discrete – item/model selection – or continuous – a real-valued budget share); **Zero-shot** – whether per-task training data, fine-tuning, or RL is required; **Dependency** – whether the items sharing the budget are independent (e.g., batched queries, weakly-coupled MDPs) or compose into a single joint outcome (the failure of one phase voids the others).

Table 3. Per-axis comparison of ZEBRA with prior budgeted-reasoning, multi-agent cost-control, budget-aware agent-control, and resource-allocation methods. Superscripts mark terms defined in the legend below the table.

Method	Persp.	Decision (params)	Zero-shot	Dependency
Inference-time token budget allocation				
TALE (Han et al., 2025)	Provider	Discrete: per-query token budget, injected as prompt constraint	Both[a]	Independent queries
Pred. Scheduling (Brown et al., 2025)	Provider	Discrete: greedy allocation of 16-token windows across queries	No (MLP on hidden states)	Independent queries
ROI-Reasoning (Zhao et al., 2026)	Provider	Discrete: per-query token budget + skip decision (ordered-subset MCKP)	No (RL fine-tuning)	Independent queries
HBPO (Lyu et al., 2025)	Provider	Discrete: per-query hierarchical token-budget tier	No (RL fine-tuning)	Independent queries
Multi-agent cost management and routing				
FrugalGPT (Chen et al., 2024)	Provider	Discrete: which model in a per-query cascade	No (trained cascade)	Independent queries
CoRL (Jin et al., 2025)	Provider	Discrete: which expert LLM to invoke per query	No (RL controller)	Independent queries
BAMAS (Yang et al., 2026)	Orchestr.	Discrete: pre-execution choice of agent team (via ILP[b]) + collaboration topology[c] (via RL)	No (ILP + RL)	Dependent (chosen agents compose a pipeline)
DAAO (Su et al., 2026)	Orchestr.	Discrete: per-query workflow structure – which agents, in what arrangement	No (VAE difficulty estimator)	Dependent (chosen agents compose a workflow)
Budget-aware agent control (inside a running agent)				
BATS (Liu et al., 2025)	Orchestr.	Discrete per-step choice inside a single agent: "dig deeper" (verify/elaborate) vs. "pivot" (try a new branch)	Yes (plug-in)	Sequential single-agent
INTENT (Liu et al., 2026)	Orchestr.	Discrete: next tool call inside a single agent's online planning loop	No (hierarchical world model[d])	Sequential single-agent
BAVT (Li et al., 2026)	Orchestr.	Discrete: which node of the single-agent multi-hop reasoning tree to expand next[e]	Yes (heuristic[f])	Sequential single-agent
Resource allocation theory and knapsack methods				
Budget MDPs (Boutilier & Lu, 2016)	Neither (Theory)	Discrete: budget chunks across weakly-coupled[g] MDPs via MCKP over piecewise-linear concave utility curves	No (known BMDP[h]; offline DP)	Independent (weak coupling)
Knapsack RL (Li et al., 2025)	Neither (Training)	Discrete: RL rollout distribution across training tasks	No (training-time)	Independent training batches
ZEBRA (ours)	**Orchestr.**	**Continuous** per-phase monetary budget across the composing phases of a multi-agent pipeline; nonlinear continuous knapsack solved via water-filling on the Lagrange multiplier	**Yes** – LLM-elicited per-phase curves; no fine-tuning, RL, or historical data	**Dependent** – multiplicative-product aggregation; one phase's failure voids the joint outcome

Legend. [a] Both: TALE has a prompting variant (zero-shot) and a fine-tuning variant (not zero-shot). [b] ILP = Integer Linear Programming, used to choose which agents form the team subject to a budget constraint. [c] Topology = the message-passing graph specifying which agents communicate with which (the structure of the multi-agent collaboration). [d] Intention-aware hierarchical world model: a learned model trained on past tool-use episodes that predicts future tool calls, costs, and outcomes; planning is done against this learned model. [e] The reasoning tree models multi-hop reasoning as a tree where each node is a reasoning step; the search decides which child node to expand next. [f] Heuristic: BAVT rescales each node's value by the remaining-budget ratio (used as a scaling exponent) – no learned component. [g] Weakly-coupled MDPs share only the budget constraint; their state, action, and reward dynamics are otherwise independent. [h] The method requires a fully specified Budgeted MDP (BMDP) model whose per-MDP value functions are precomputed offline via dynamic programming.

A.2. Economic mechanisms in multi-agent systems

(Hua et al., 2025) apply Shapley values to credit assignment and pricing negotiation among self-interested agents, demonstrating that principled algorithmic guidance can meaningfully improve coordination outcomes in multi-agent systems.

A.3. Detailed per-paper descriptions (full prose, moved from Section 2)

This subsection preserves the full per-paper exposition that appeared in earlier drafts of Section 2; the body retains only a citation-list summary.

Inference-time token budget allocation. A large body of work studies how inference operators can redistribute a fixed token budget across independent user queries to maximize aggregate accuracy. Difficulty estimation is a common thread: (Han et al., 2025) estimate a per-query token budget via prompting or fine-tuning and inject it as a reasoning constraint (TALE); (Brown et al., 2025) train an MLP on transformer hidden states to predict early-stopping probabilities, then greedily allocate 16-token windows to whichever query yields the highest marginal gain. Methods that require RL fine-tuning go further by baking budget awareness directly into model weights: (Zhao et al., 2026) train a model to predict token need and skip low-ROI queries (ROI-Reasoning); (Lyu et al., 2025) partition the token budget into hierarchical tiers and learn per-tier reward structures (HBPO). All four take the *provider* perspective, operate on batches of *independent* queries, and pick a *discrete* per-query token budget; most also require task-specific training data or model fine-tuning. ZEBRA differs on every axis (Table 3): it is an *orchestration*-side controller that splits a *continuous* monetary budget across the *dependent* phases of a single multi-agent pipeline, zero-shot.

Multi-agent cost management and routing. A parallel line reduces cost through discrete model selection and agent routing rather than continuous budget allocation: (Chen et al., 2024) build an LLM cascade that escalates to expensive models only when needed (FrugalGPT); (Jin et al., 2025) train an RL controller to pick which expert LLM to invoke per query under a budget (CoRL); (Yang et al., 2026) combine ILP-based agent selection with RL-fixed collaboration topology (BAMAS); (Su et al., 2026) use a VAE difficulty estimator to assemble per-query workflows (DAAO); (Kandogan et al., 2025) treat budget as a constraint in enterprise pipeline design; and (Amayuelas et al., 2025) find that upfront worker-model selection beats reactive orchestration. In every case the decision variable is *discrete and combinatorial* and the controller is trained (Table 3).

Budget-aware agent control. A concurrent line studies budget-aware control at inference time *inside* a single agent. BATS (Liu et al., 2025) pairs a Budget Tracker plug-in with a per-step "dig deeper vs. pivot" decision under a remaining tool-call budget; INTENT (Liu et al., 2026) trains an intention-aware hierarchical world model that risk-calibrates next-tool calls online for a single agent under a strict monetary budget; BAVT (Li et al., 2026) models multi-hop reasoning as a dynamic value tree and uses the remaining-resource ratio as a scaling exponent on node values. All three are *step-by-step* controllers inside a single running agent. ZEBRA targets a different and complementary problem *above* the pipeline: before any agent runs, it solves a one-shot continuous knapsack across all phases. The two layers stack – ZEBRA's per-phase budgets define the caps within which a step-level controller of this kind would operate.

Resource allocation theory and knapsack methods. The knapsack problem and its variants provide the theoretical backbone for budget allocation (Katoh & Ibaraki, 1998; Kellerer et al., 2004; Boyd & Vandenberghe, 2004). Boutilier & Lu (2016) allocate a fixed budget across weakly-coupled MDPs via Multiple-Choice Knapsack (MCKP), assuming full independence between subprocesses and predating LLMs entirely. Recent LLM work has adopted knapsack formulations for related problems: (Zhao et al., 2026) use an ordered-subset MCKP across independent queries, (Brown et al., 2025) solves a greedy knapsack over fixed-size token windows, and (Li et al., 2025) models RL rollout distribution as a knapsack. All operate on independent queries or training batches rather than across the composing phases of a single running pipeline.

Hybrid LLM + knapsack-solver systems and LLM weakness on NP-hard problems. The methodological precedent closest to ZEBRA is SEER (Tonglet et al., 2023), which formulates in-context exemplar selection for HybridQA as an integer-linear-program knapsack with diversity and capacity constraints, demonstrating that wrapping an LLM system with an explicit combinatorial-optimization layer beats heuristic selection. SEER and ZEBRA share the high-level move – delegate a constrained selection/allocation step to a knapsack solver rather than to the LLM itself – but differ in scope and problem class. SEER addresses one-prompt, discrete exemplar choice over an independent candidate set; ZEBRA addresses zero-shot, continuous, orchestration-time monetary-budget allocation across the *dependent* phases of a multi-phase pipeline

whose outputs compose into a single joint outcome. The choice to delegate the allocation step at all is independently supported by Duchnowski & Pavlick (2025) (EHOP), who introduce a benchmark of NP-hard problems – including knapsack – expressed in natural language and show that frontier LLMs, including reasoning models, systematically solve textbook formulations more accurately than real-life or rule-inverted variants and exhibit high variance across surface presentations. Their finding – that current LLMs lack a robust, presentation-invariant solver for NP-hard problems – motivates ZEBRA's design.

B. Background and Method Details

B.1. Full assumptions and convexity (from Section 3.1)

We restate the full optimization problem with optional per-phase upper bounds:

$$\max_{\mathbf{x}} \sum_{i=1}^{n} f_i(x_i) \quad \text{s.t.} \quad \sum_{i=1}^{n} x_i \le B, \;\; 0 \le x_i \le u_i \;\forall i, \tag{6}$$

where u_i is an optional per-phase upper bound (e.g., a hard cap on tokens or tool calls).

We assume each f_i is non-decreasing (more budget never hurts), concave (diminishing returns), and differentiable. Under these assumptions $\sum_i f_i(x_i)$ is concave on a convex feasible set, so (6) is a convex program with no spurious local optima and with optimality fully characterized by KKT conditions (Boyd & Vandenberghe, 2004). Without a fully convex feasible set, related continuous knapsack variants can require nonconvex machinery (Bai & Cardonha, 2023).

B.2. KKT conditions and dual-search algorithm (full statement, from Section 3.2)

Introduce a Lagrange multiplier $\lambda \ge 0$ for the budget constraint and box-multipliers for $0 \le x_i \le u_i$. Because the problem is convex, the KKT conditions are necessary and sufficient (Boyd & Vandenberghe, 2004) and yield the familiar *marginal equalization* rule

$$f_i'(x_i^\star) = \lambda \quad \text{for } 0 < x_i^\star < u_i, \tag{7}$$

with the natural boundary modifications $f_i'(x_i^\star) \le \lambda$ at $x_i^\star = 0$ and $f_i'(x_i^\star) \ge \lambda$ at $x_i^\star = u_i$. Economically, λ is the shadow price of budget: budget flows to each phase until all *active* phases have equal marginal utility, matching the water-filling solution from communication systems (Boyd & Vandenberghe, 2004).

Defining the per-phase response $x_i(\lambda) = \arg\max_{0 \le x \le u_i} \big(f_i(x) - \lambda x\big)$, the total allocation $S(\lambda) = \sum_i x_i(\lambda)$ is non-increasing in λ, so $\lambda^\star$ can be found by bisection in $O(n \log \frac{1}{\varepsilon})$ time, matching the classical resource-allocation result (Katoh & Ibaraki, 1998; Ibaraki & Katoh, 1988).

From products to sums. A weighted product $\prod_i h_i(x_i)^{w_i}$ with positive concave h_i reduces, via log, to $\sum_i w_i \log h_i(x_i)$, again a separable concave program solvable by the same dual-search machinery. ZEBRA uses this transformation to handle all multiplicative and offset objectives of Section 4 with one solver; full derivation in Appendix C.

B.3. Saturating-exponential derivatives (from Section 4.1)

The first and second derivatives of the saturating exponential $f_i(x) = a_i(1 - e^{-b_i x})$ are

$$f_i'(x) = a_i b_i e^{-b_i x}, \qquad f_i''(x) = -a_i b_i^2 e^{-b_i x} \le 0, \tag{8}$$

which establish that f_i is monotone non-decreasing and concave for $a_i, b_i > 0$.

B.4. Two-point curve fitting (full derivation, from Section 4.1)

Rather than asking the controller for abstract curve parameters, we elicit two intuitive operating points per phase. The controller estimates: (i) $n_{\text{basic},i}$, the total output tokens the phase needs to reach roughly 50% of its potential quality, and (ii) $n_{\text{great},i}$, the total output tokens to reach roughly 90% (named `tokens_basic` and `tokens_great` in the controller prompt; App. G.4). These correspond to the equations $f_i(n_{\text{basic}}) = 0.5\, a_i$ and $f_i(n_{\text{great}}) = 0.9\, a_i$, yielding two estimates:

$$b_{\text{basic}} = \frac{\ln 2}{n_{\text{basic}}}, \qquad b_{\text{great}} = \frac{\ln 10}{n_{\text{great}}}. \tag{9}$$

We combine these as $b_i = \sqrt{b_{\text{basic}} \cdot b_{\text{great}}}$. Because the global budget B is monetary (USD), we convert each phase's token operating points to USD via that phase's model pricing – which differs across phases (e.g., `gpt-4o` for `refine` vs. `gpt-4o-mini` for the others) – before computing b_i, so that b_i has units of inverse-USD and the per-phase response $x_i(\lambda)$ in Section 4.2.1 is in USD. The equations above are unit-agnostic: the same expression for b holds whether n is in tokens or in dollars; the conversion just rescales b by the per-phase USD-per-token rate. In addition to b_i, the controller also estimates $a_i \in (0, 1]$, the quality ceiling of each phase. (A propagation-weighted variant that reuses a_i as a per-phase weight on the multiplicative offset objective is defined and evaluated in Appendix D.12.)

B.5. Multiplicative offset closed form (from Section 4.2.2)

Taking logs (Appendix C) reduces (5) to the separable concave program $\max \sum_i \log g_i(x_i)$, solvable by the same dual search. The KKT condition $g_i'(x_i)/g_i(x_i) = \lambda$ yields the closed-form per-phase response

$$x_i(\lambda) \;=\; \max\Big(0,\; \tfrac{1}{b_i} \ln \tfrac{a_i(b_i+\lambda)}{\lambda}\Big). \tag{10}$$

Phase i is starved exactly when its *log-marginal at zero*, $g_i'(0)/g_i(0) = a_i b_i/(1 - a_i)$, falls below the shadow price λ. The form is parallel to the additive case (marginal at zero $f_i'(0) = a_i b_i$ vs. λ): in both objectives, a phase is starved when its initial return on budget is too low to beat the going price. The shift from f' to g'/g is just the chain rule applied to $\log g_i$, since the multiplicative objective is a sum of *logs* after the transformation.

C. Logarithmic Transformation Derivation

A natural extension of the additive formulation in Section 3.1 replaces the sum with a product:

$$\max_{\mathbf{x}} \quad \prod_{i=1}^{n} h_i(x_i) \quad \text{s.t.} \sum_{i=1}^{n} x_i \le B,\; x_i \ge 0, \tag{11}$$

where each h_i is positive, concave, and non-decreasing. Because the logarithm is strictly increasing, the product objective shares its maximizer with the log-sum:

$$\max_{\mathbf{x}} \quad \sum_{i=1}^{n} \log h_i(x_i) \quad \text{s.t.} \sum_{i=1}^{n} x_i \le B,\; x_i \ge 0. \tag{12}$$

When each h_i is positive and log-concave (which is guaranteed when h_i is concave and positive), each term $\log h_i(x_i)$ is concave, so (12) is again a separable concave program with a linear constraint. The water-filling machinery of Section 3.2 applies directly: one searches for a Lagrange multiplier λ such that the log-marginals $\frac{d}{dx} \log h_i(x_i) = h_i'(x_i)/h_i(x_i)$ are equalized across active phases.

More generally, a *weighted* product $\prod_i h_i(x_i)^{w_i}$ with $w_i > 0$ transforms to $\sum_i w_i \log h_i(x_i)$, and the optimality condition becomes $w_i \cdot h_i'(x_i)/h_i(x_i) = \lambda$ for all active phases. This observation is central to ZEBRA: it allows us to solve multiplicative, propagation, and offset objectives (Section 4) using the same dual-search template as the additive case, differing only in the closed-form expression for $x_i(\lambda)$ at each step.

D. Additional Experiment Results

D.1. Pipeline phases

See Table 4 for details.

D.2. Benchmark construction details

Raw pool. We start from the APPS benchmark (Hendrycks et al., 2021) at the *interview* difficulty level. To keep scoring reliable we restrict to tasks with enough test cases to distinguish near-solutions from fully-correct ones. We use two ranges of test-suite size: tasks with at least 100 tests (103 tasks, `apps_tasks/`) and tasks with 50–99 tests (338 tasks, `apps_tasks_50/`), extracted with `convert_apps_tasks_50.py` in our code release.

Phase	Model	Purpose
`plan`	`gpt-4o-mini`	Produce a high-level plan for solving the task.
`decompose`	`gpt-4o-mini`	Decompose the plan into implementable subtasks.
`implement`	`gpt-4o-mini`	Write solution code across subtasks.
`refine`	`gpt-4o`	Review→revise loop: identify real bugs via spec comparison, then fix flagged issues only. Up to 3 iterations.

Table 4. Workflow phases shared by all strategies. The refine phase uses a stronger model (`gpt-4o`) for higher-quality code review; all other phases use `gpt-4o-mini`.

Screening for stability. We screen every task in both pools with 30 independent `no_budget` runs of the same pipeline used for the benchmark, using the same models. For each task we record the mean and coefficient of variation of cost and of score across the 30 runs. A task is retained if $\mathrm{CV}_{\text{cost}} < 0.35$ and its cost variance is non-zero (trivially solved tasks are excluded). This retains 49 of 103 tasks in the 100+-test pool and 170 of 338 tasks in the 50–99-test pool. We save the per-task means.

Balancing across tiers. We assign each retained task a difficulty tier from its mean no-budget score $\bar{s}$: *easy* ($\bar{s} \geq 0.8$), *medium* ($0.4 \leq \bar{s} < 0.8$), *hard* ($\bar{s} < 0.4$). The 49-task pool distributes as 22 easy / 20 medium / 7 hard, which is too hard-light for a per-tier analysis. We therefore sample from the 50–99-test stable pool to fill each tier to 50 tasks:

Tier	Pool 1 (100+)	Pool 2 (50–99)	Total
Easy	22	28	50
Medium	20	30	50
Hard	7	43	50
Total	49	101	150

Job counts. The main benchmark comprises $150 \times 2 \times 3 \times 15 = 13{,}500$ runs for the main ($\alpha \in \{0.5, 0.8\}$) portion plus $50 \times 3 \times 15 = 2{,}250$ runs for the $\alpha = 0.3$ easy-tasks portion, on top of the $150 \times 30 = 4{,}500$ `no_budget` screening runs.

D.3. Why we drop the budget-aware a-rescaling

An earlier draft of ZEBRA included a budget-aware rescaling of the quality ceilings, $a'_i = a_i^{\kappa}$ with $\kappa = \max(1,\ 1 + 2\ln\frac{4}{c})$, intended to let the offset objectives meaningfully starve phases when the budget is tight. We ran the full 150-task benchmark with and without rescaling on `prop_offset` and `mult_offset`; the no-rescale ($\kappa \equiv 1$) variants matched or beat the rescaling variants on every aggregate metric at both $\alpha \in \{0.5, 0.8\}$. The rescaling step is therefore dropped throughout this paper, and all results in Section 5.2– 5.6 use the simpler $\kappa \equiv 1$ formulation. Removing the rescaling has the additional benefit that the curves estimated by the controller are used as-is, without a budget-dependent post-hoc transformation, which makes the controller's output easier to inspect.

D.4. Paired Wilcoxon details

For every comparison in Sections 5.2–5.5, we run paired Wilcoxon signed-rank tests on three per-task metrics (mean score, median score, success rate), pairing each ZEBRA variant against LLM-direct on the same tasks. Below we report the mean-score test results in full; the median-score and success-rate tests are qualitatively similar (always significant where the mean test is significant).

The 18 tests in Table 5 form a single corrected family. We report the raw two-sided p-value (p_{raw}) alongside the Benjamini–Hochberg adjusted p-value (p_{BH}) at $q = 0.05$ over $m = 18$ tests; the significance markers in the body and in this table reflect p_{BH}. Of the 13 raw-significant cells, all 13 remain significant under BH-FDR.

D.5. API model snapshots and sampling parameters

All experiments use OpenAI's Chat Completions API. To ensure reproducibility, we report the exact model snapshots resolved by each alias during our experiment window:

Condition	Strategy	Mean-diff	W/L/T	p_{raw}	p_{BH}	Sig.
$\alpha = 0.5$, all ($n = 150$)	zebra-additive	$+0.0220$	$74/38/38$	1.27×10^{-3}	4.59×10^{-3}	**
$\alpha = 0.5$, all ($n = 150$)	zebra-mult_offset	$+0.0359$	$79/34/37$	2.63×10^{-7}	4.74×10^{-6}	***
$\alpha = 0.8$, all ($n = 150$)	zebra-additive	$+0.0216$	$77/38/35$	8.36×10^{-5}	5.01×10^{-4}	***
$\alpha = 0.8$, all ($n = 150$)	zebra-mult_offset	$+0.0187$	$75/39/36$	6.95×10^{-4}	3.13×10^{-3}	**
$\alpha = 0.5$, easy ($n = 50$)	zebra-additive	-0.0110	$11/10/29$	0.1909	0.2291	ns
$\alpha = 0.5$, easy ($n = 50$)	zebra-mult_offset	$+0.0060$	$13/7/30$	0.1790	0.2291	ns
$\alpha = 0.5$, medium ($n = 50$)	zebra-additive	$+0.0491$	$32/14/4$	2.48×10^{-3}	7.43×10^{-3}	**
$\alpha = 0.5$, medium ($n = 50$)	zebra-mult_offset	$+0.0673$	$34/12/4$	4.09×10^{-5}	3.68×10^{-4}	***
$\alpha = 0.5$, hard ($n = 50$)	zebra-additive	$+0.0278$	$31/14/5$	4.22×10^{-3}	7.95×10^{-3}	**
$\alpha = 0.5$, hard ($n = 50$)	zebra-mult_offset	$+0.0344$	$32/15/3$	4.42×10^{-3}	7.95×10^{-3}	**
$\alpha = 0.8$, easy ($n = 50$)	zebra-additive	$+0.0051$	$11/13/26$	0.6071	0.6428	ns
$\alpha = 0.8$, easy ($n = 50$)	zebra-mult_offset	$+0.0076$	$15/9/26$	0.2652	0.2983	ns
$\alpha = 0.8$, medium ($n = 50$)	zebra-additive	$+0.0418$	$34/12/4$	4.27×10^{-3}	7.95×10^{-3}	**
$\alpha = 0.8$, medium ($n = 50$)	zebra-mult_offset	$+0.0343$	$32/17/1$	0.0219	0.0303	*
$\alpha = 0.8$, hard ($n = 50$)	zebra-additive	$+0.0179$	$32/13/5$	4.07×10^{-3}	7.95×10^{-3}	**
$\alpha = 0.8$, hard ($n = 50$)	zebra-mult_offset	$+0.0142$	$28/13/9$	0.0187	0.0296	*
$\alpha = 0.3$, easy ($n = 50$)	zebra-additive	$+0.0154$	$19/8/23$	0.0197	0.0296	*
$\alpha = 0.3$, easy ($n = 50$)	zebra-mult_offset	$+0.0026$	$11/15/24$	0.8291	0.8291	ns

Table 5. **Paired Wilcoxon signed-rank tests on per-task mean score, ZEBRA variants vs LLM-direct.** Two-sided test. p_{raw} is the unadjusted Wilcoxon p-value; p_{BH} is the Benjamini–Hochberg adjusted p-value over the $m = 18$ tests in this family. Sig. codes use p_{BH}: $^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$, ns = not significant. W/L/T = number of tasks where ZEBRA wins / loses / ties LLM-direct on per-task mean.

Alias	Snapshot	Used for
`gpt-4o-mini`	`gpt-4o-mini-2024-07-18`	`plan`, `decompose`, `implement`
`gpt-4o`	`gpt-4o-2024-08-06`	`refine`, controller

Snapshots were confirmed via OpenAI's published model lifecycle documentation. Sampling temperature is set per call type to balance determinism (controller decisions, code generation) against diversity (planning):

Call type	`temperature`
`plan`	0.7
`decompose`	0.5
`implement`, `refine`, `review`	0.3
Controller (ZEBRA, LLM-direct allocator)	0.3

`top_p` is left at the SDK default (1.0). Run-to-run variability is addressed by averaging over 15 independent runs per (task, α, strategy) condition.

D.6. Ablations on the allocation step

We hold the rest of the pipeline fixed (same phases, same models, same total budget, same prompts) and replace only ZEBRA's allocator, to isolate the contribution of (i) the knapsack solver and (ii) the phase utility curves themselves. Both ablations use `zebra-mult_offset` (no rescale) as the reference strategy. Results are summarized in Table 6.

Ablation 1: LLM allocator on the same fitted curves. We give the controller (`gpt-4o`, same model used by ZEBRA's estimation step) the same fitted per-phase curves $\{(a_i, b_i)\}_{i=1}^{n}$ that ZEBRA estimates and ask it to output an allocation directly. This isolates the contribution of the *optimization step*: any difference between this ablation and ZEBRA must come from the controller's ad-hoc allocation versus the knapsack solution on identical inputs. The controller's allocations score -4.20 points worse at $\alpha = 0.5$ (0.4977 vs. 0.5397) and -4.31 points worse at $\alpha = 0.8$ (0.5146 vs. 0.5576), both significant at $p < 10^{-4}$ ($n = 150$ paired Wilcoxon signed-rank).

α	Allocator (curves used?)	Mean	Succ%	ΔMean	p
0.5	ZEBRA knapsack (yes)	**0.5397**	**54.1%**	—	—
	LLM, same curves	0.4977	48.4%	−0.0420	$<10^{-4}$***
	Uniform 25/25/25/25	0.4883	47.2%	−0.0514	$<10^{-4}$***
0.8	ZEBRA knapsack (yes)	**0.5576**	**55.2%**	—	—
	LLM, same curves	0.5146	50.5%	−0.0431	$<10^{-4}$***
	Uniform 25/25/25/25	0.4985	48.2%	−0.0592	$<10^{-4}$***

Table 6. **Ablations on the allocation step.** Reference is `zebra-mult_offset`. Both ablations hold the rest of the pipeline fixed (same phases, same models, same budget, same prompts) and replace only ZEBRA's allocator. Ablation 1 hands the controller the same fitted phase curves and asks it to output an allocation directly, isolating the contribution of the knapsack solver. Ablation 2 ignores curves entirely and uses a flat 25% split per phase, isolating the contribution of curve-aware allocation. Both ablations are significantly worse than ZEBRA at both budget levels (paired Wilcoxon signed-rank one-sided $p < 10^{-4}$, $n = 150$).

Ablation 2: Uniform $25/25/25/25$ split. We discard the curves entirely and use a flat per-phase split. This isolates the contribution of *curve-aware allocation*: the gap to ZEBRA reflects the value of estimating phase utilities at all. The uniform split scores -5.14 points worse at $\alpha = 0.5$ (0.4883 vs. 0.5397) and -5.92 points worse at $\alpha = 0.8$ (0.4985 vs. 0.5576), again significant at $p < 10^{-4}$.

Ablation 3: Chain-of-thought prompting on LLM-direct. A natural concern is whether LLM-direct's underperformance in Table 1 simply reflects weak prompting – if the controller were asked to think step-by-step before allocating, would the gap close? We test this directly with LLM-CoT, identical to LLM-direct but with a chain-of-thought scaffold (the prompt is reproduced in Appendix G.4). We run the full 150-task benchmark at $\alpha = 0.5$. The result is unambiguous: **CoT does not help.** LLM-CoT scores 0.4968 overall vs. 0.5038 for plain LLM-direct ($\Delta = -0.0070$, paired Wilcoxon $p_{\text{raw}} = 0.054$, n.s., $n = 150$); the trend is in the wrong direction (W/L/T = 42/75/33), not the right one. ZEBRA still beats CoT by a wide margin: `zebra-mult_offset` scores +0.0429 over LLM-CoT (W/L/T = 90/29/31, $p_{\text{raw}} < 10^{-4}$) and `zebra-additive` scores +0.0290 over LLM-CoT ($p_{\text{raw}} = 0.0006$). The gap is therefore not a prompting artefact; the LLM controller cannot match the knapsack solution even with explicit reasoning steps.

Takeaway. All three ingredients matter independently of one another. Removing the solver costs ∼4 points; removing the curves costs ∼5–6 points; adding CoT to the LLM-direct controller does not recover any of the gap. Together these results explain why LLM-direct in Tables 1–2 underperforms even when it has access to the same controller model: even given the fitted curves *and* explicit reasoning steps, the controller cannot match the knapsack solution.

D.7. Empirical validation of controller phase utility curves

The ZEBRA solver consumes per-phase utility curves $u_i(x) = a_i(1 - e^{-b_i x})$ produced by the controller. The ablations in Appendix D.6 show that removing those curves (uniform split) is the single most damaging ingredient to remove, but they do not directly check whether the controller's *shape* prediction matches the empirical phase response. This appendix reports a post-hoc calibration of the controller's curve for the `refine` phase against the empirical Δscore it produces in our run logs (Figure 3). We focus on `refine` because, unlike the upstream phases whose intermediate outputs are not independently scored, `refine`'s contribution can be cleanly isolated from the difference between the implement-stage candidate and the final selected candidate.

Data extraction. For each run log we parse, via regex, the controller-printed line `Controller curves (...): {...}` to recover (a_i, b_i) for each phase. The `refine` curve is fit by the controller in token space; we replace its $b_{\texttt{refine}}$ with the cost-adjusted value reported by the `Refine cost adjustment:...adjusted b=X` line, which divides the token-space b by ∼16.7× (the `gpt-4o-mini` → `gpt-4o` cost ratio) so that the curve is expressed in dollars rather than tokens. We then read three quantities from the same log: the implement-stage score (`[refine] candidate 0 (implement): score=X`), the final selected score (`[refine] Selected candidate N ... with score=X`; if the log records `Final code = implement output` the final score equals the implement score), and the realized refine spend (`[refine] Spent=$X`). The empirical refine contribution is Δscore = final − implement.

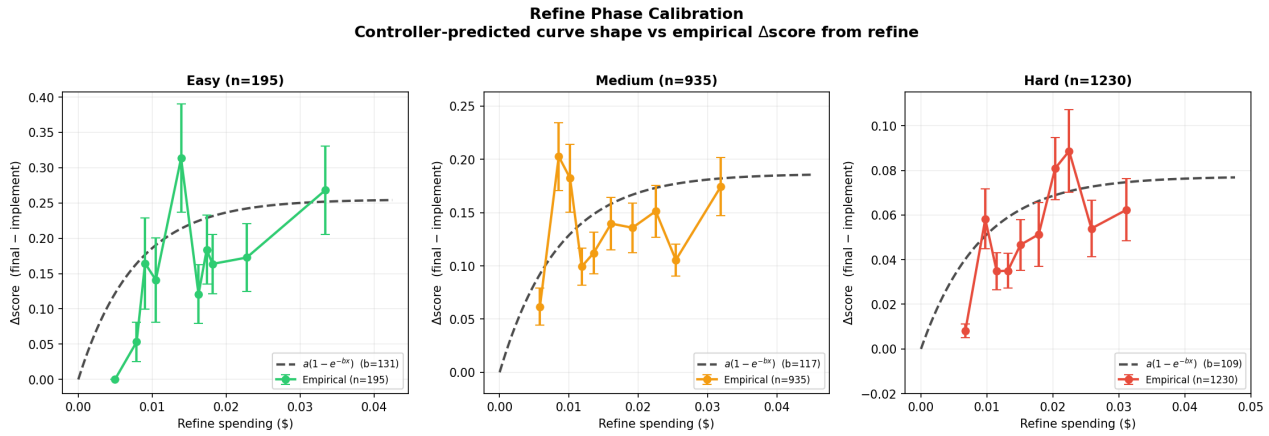

Figure 3. **Refine-phase calibration: controller-predicted curve shape vs. empirical Δscore from refine.** Empirical points are mean $\Delta\text{score} = \text{final} - \text{implement}$ within ten equal-count quantile bins of realized refine spend, with standard-error bars; runs are capped at 30 per task before binning. The dashed curve is the controller's predicted shape $a(1 - e^{-bx})$ at the tier-average cost-adjusted b, normalized so its plateau matches the empirical plateau (top-three-bin mean).

Difficulty assignment and balancing. Tasks are tier-labelled from `nb_costs.json` (the no-budget screening means used in Appendix D.2): easy ($\bar{s} \geq 0.8$), medium ($0.4 \leq \bar{s} < 0.8$), hard ($\bar{s} < 0.4$). Because some tasks have many more runs than others (up to 90 vs. as few as 1), we cap each task at 30 runs by random sampling (seed $= 42$) before pooling, so that no single task dominates a tier.

Binning and curve overlay. Within each tier, runs are sorted by realized refine spend and split into ten equal-count quantile bins. We plot each bin's mean Δscore with its standard error (empirical points). The controller's predicted shape is overlaid as $a(1 - e^{-bx})$ using the tier-average a and cost-adjusted b, with a rescaled so that the curve's plateau matches the empirical plateau (mean of the top three bins). This rescaling absorbs any tier-level offset in a and tests only the *shape* (saturation rate) determined by b.

Result. Across all three tiers the empirical points lie close to the controller's predicted curve, with mild over-prediction in the lowest-spend bins and good agreement past ∼\$0.015 refine spend. The fitted saturation rates ($b \approx 109$–131 across tiers) reproduce the empirical knee location, supporting the use of the estimated curves as inputs to the knapsack solver in Section 4.

D.8. Per-task scatter view and binding-budget intuition

The per-tier structure in Section 5.3 is consistent with the intuition behind ZEBRA: allocation quality only matters when the budget is binding. When the task is easy, any reasonable split works; when the task is hard or the budget is tight, the choice of split becomes the main lever. Figure 4 visualises this per-task: each point is a task, plotted as the best ZEBRA variant's mean score (vertical) versus LLM-direct's mean score (horizontal), with the diagonal indicating parity. Easy tasks cluster on the diagonal, while medium and especially hard tasks drift above it – the drift away from the diagonal grows with task difficulty.

D.9. Detailed budget-pressure results

The very-tight ($\alpha = 0.3$) regime on the 50 easy tasks (Section 5.4) is reported in full in Table 7. `zebra-additive` recovers 96.1% of the no-budget score versus 94.5% for LLM-direct, beating LLM-direct on per-task mean in 19/50 tasks (loses 8, ties 23; paired Wilcoxon $p_{\text{raw}} = 0.0197$, $p_{\text{BH}} = 0.030$) and winning outright on 38/50 tasks. Figure 5 visualises NB-retention vs. budget tightness across all conditions.

Per-phase allocation at $\alpha = 0.3$. Table 8 reports the mean per-phase budget fractions for the very-tight regime, supplementing Table 9 (which covers $\alpha = 0.5$ and $\alpha = 0.8$). Both ZEBRA variants compress `refine` further at $\alpha = 0.3$ (10.7% for `additive`, 19.3% for `mult_offset`) than at $\alpha = 0.5$ on the same easy tier (30.5%, 45.1%); LLM-direct's `refine`

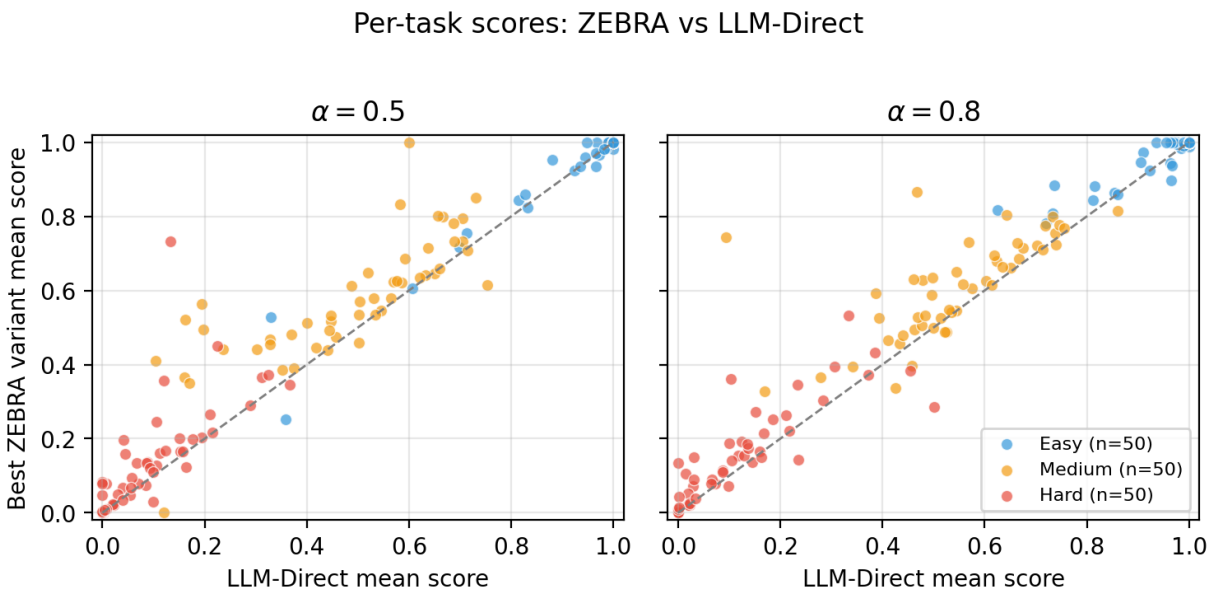

Figure 4. **Per-task scores: best ZEBRA variant vs LLM-direct.** Each dot is a task; y-axis is the best ZEBRA variant's per-task mean score, x-axis is LLM-direct's. Dashed line is $y = x$. Points above the diagonal are tasks where ZEBRA wins. Easy tasks (blue) cluster near the diagonal in the upper right; the per-task gap grows as tasks get harder, with a visible swarm of red points well above the diagonal at low LLM scores – those are tasks where LLM-direct scores near zero and ZEBRA recovers some signal.

Strategy	Mean	Succ%	NB ret.	vs LLM (W/L/T)
`zebra-additive`	**0.927***	**94.7**	**96.1**	**19/8/23**
`zebra-mult_offset`	0.915	92.9	94.8	11/15/24
`llm`	0.912	93.1	94.5	—

Table 7. **Very-tight budget on easy tasks** ($\alpha = 0.3$, $n = 50$)**.** `zebra-additive` beats LLM-direct on per-task mean score (paired Wilcoxon $p_{\text{raw}} = 0.0197$, $p_{\text{BH}} = 0.0296$, marked *). `zebra-mult_offset` essentially ties LLM-direct ($p_{\text{raw}} = 0.8291$, ns), with W/L/T = 11/15/24 on per-task mean. Bold = best in column. Asterisks reflect BH-adjusted p-values from the 18-test APPS main family (Appendix D.4).

share is essentially unchanged across budgets (47.3% at $\alpha = 0.3$ vs. 45.8% at $\alpha = 0.5$).

D.10. Per-phase allocation breakdown

Table 9 reports the average per-phase budget fractions referenced in Section 5.5, broken down by tier and budget multiplier; Figure 6 visualises the same distributions.

D.11. Per-task win counts across objectives

Table 10 accompanies the discussion of "Additive vs. `mult_offset`: when does the product aggregator help?" in Section 5.5: it reports per-task win counts for all four strategies (`additive`, `mult_offset`, `prop_offset`, and LLM-direct) across tiers and budget multipliers. The `prop_offset` column is included for completeness; that variant and its head-to-head with `mult_offset` are discussed in Appendix D.12.

D.12. Propagation-weighted variant: definition and comparison to `mult_offset`

The body of the paper compares two ZEBRA objectives, `additive` (Eq. 3) and `mult_offset` (Eq. 5). Here we define a third, propagation-weighted variant – which adds per-phase dependency weights on top of `mult_offset` – and report its empirical comparison against `mult_offset`. The headline finding: the additional per-phase weighting does *not* reliably move the per-task score over the symmetric `mult_offset`, so we leave it out of the body and report it here for

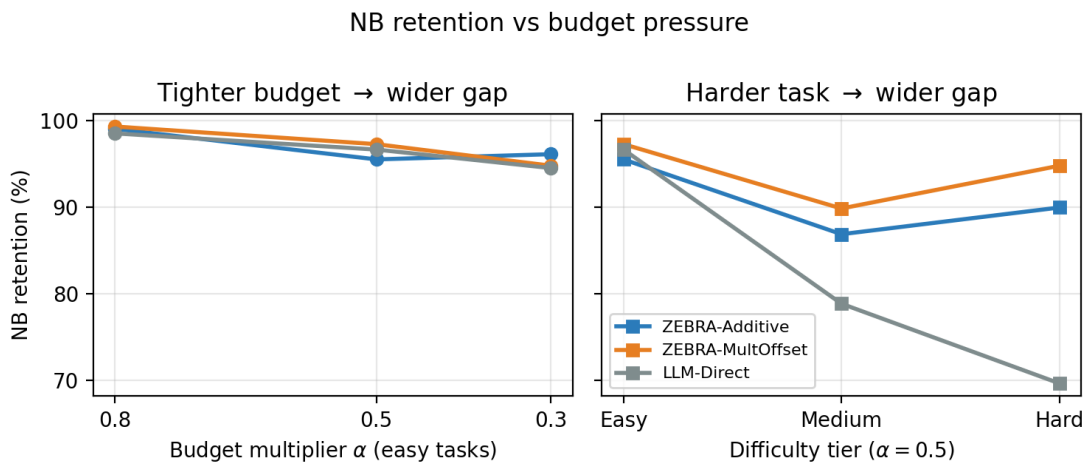

Figure 5. **NB retention vs budget tightness.** Fraction of unconstrained quality recovered by each strategy as the budget multiplier α tightens. The ZEBRA-vs-LLM gap is essentially zero at $\alpha = 0.8$ on easy tasks and grows as either the budget tightens (easy: $\alpha = 0.5 \rightarrow 0.3$) or the tier hardens ($\alpha = 0.5$, easy $\rightarrow$ medium $\rightarrow$ hard).

Strategy	`plan`	`decomp`	`impl`	`refine`
zebra-additive	22.4	26.4	40.4	**10.7**
zebra-mult_offset	20.3	23.7	36.8	**19.3**
llm	12.3	14.2	26.2	**47.3**

Table 8. **Per-phase budget fractions at $\alpha = 0.3$ (very-tight, easy tasks, $n = 50$, 15 runs per cell).** Mean allocation (%) across the four pipeline phases. **Both ZEBRA variants compress `refine` further as the budget tightens** – to 10.7% (`additive`) and 19.3% (`mult_offset`) at $\alpha = 0.3$, down from their easy-tier $\alpha = 0.5$ shares of 30.5% and 45.1% respectively (Table 9). LLM-direct keeps `refine` near 47%, essentially unchanged from its $\alpha = 0.5$ easy-tier share of 45.8%. This empirically substantiates the behavioral claim in Section 5.5.

completeness.

Definition. The multiplicative objective in Eq. (5) treats all phases symmetrically. To allow per-phase emphasis – motivated by the intuition that early phases (e.g., `plan`) cascade more strongly into downstream phases than late phases – we introduce dependency weights $w_i > 0$ and maximize:

$$\max_{\mathbf{x}} \quad \prod_{i=1}^{n} \left(1 - a_i\, e^{-b_i x_i}\right)^{w_i} \quad \text{s.t.} \sum_i x_i \leq B,\ x_i \geq 0. \tag{13}$$

Via the log transformation (Appendix C), this becomes $\max \sum_i w_i \log(1 - a_i e^{-b_i x_i})$ and the per-phase allocation is:

$$x_i(\lambda) = \max\left(0,\ \frac{1}{b_i} \ln \frac{a_i(w_i b_i + \lambda)}{\lambda}\right). \tag{14}$$

Phase i receives zero budget when $\lambda \geq a_i w_i b_i/(1 - a_i)$. The weight w_i is set to the quality ceiling a_i already estimated by the controller. Phases with higher ceilings which contribute more to the product when they succeed therefore receive proportionally more weight, while `prop_offset` reduces to `mult_offset` when $w_i \equiv 1$ (uniform weighting).

Empirical comparison: `prop_offset` vs. `mult_offset`. We run `prop_offset` on the same 150-task benchmark with the same 15 runs per cell. Table 11 reports the side-by-side comparison and a paired Wilcoxon signed-rank test on per-task mean score against `mult_offset` (the natural $w_i \equiv 1$ control). The picture is consistent across both budget levels: differences are small, do not have a consistent sign across cells, and are not statistically significant at the overall level.

Allocations are nearly identical. The mean per-phase budget shares of `prop_offset` and `mult_offset` differ by at most ∼2 percentage points on any phase at either budget level (e.g., at $\alpha = 0.5$: `plan` 12.2% vs. 11.7%; `decompose`

Tier	Strategy	Mean allocation (%), $\alpha = 0.5$				Mean allocation (%), $\alpha = 0.8$			
		plan	decomp	impl	refine	plan	decomp	impl	refine
Easy	zebra-additive	16.6	20.3	32.6	30.5	11.6	14.3	23.2	50.9
	zebra-mult_offset	15.5	14.5	24.9	45.1	12.2	11.3	19.8	56.7
	llm	13.7	14.2	26.3	45.8	13.7	13.2	26.7	46.4
Medium + Hard	zebra-additive	8.7	10.8	19.8	60.7	6.2	7.8	14.6	71.3
	zebra-mult_offset	9.9	8.3	16.2	65.7	8.2	6.7	13.7	71.3
	llm	11.9	12.3	25.2	50.6	12.3	12.7	24.1	50.9

Table 9. **Where each strategy spends its budget.** Mean per-phase budget fractions at $\alpha = 0.5$ and $\alpha = 0.8$, grouped by difficulty tier. **Both ZEBRA variants reallocate across tiers**: on easy tasks they assign `refine` only 30–57% of the budget, while on medium+hard tasks both variants pull strongly toward `refine` (60–71%). LLM-direct uses essentially the same ∼25% `implement` / ∼50% `refine` split across tiers and budgets. The within-strategy shift is largest for `additive` (a 30-point drop in `refine` share between tiers at $\alpha = 0.5$). This adaptation is the proximate cause of the score gaps in Table 2.

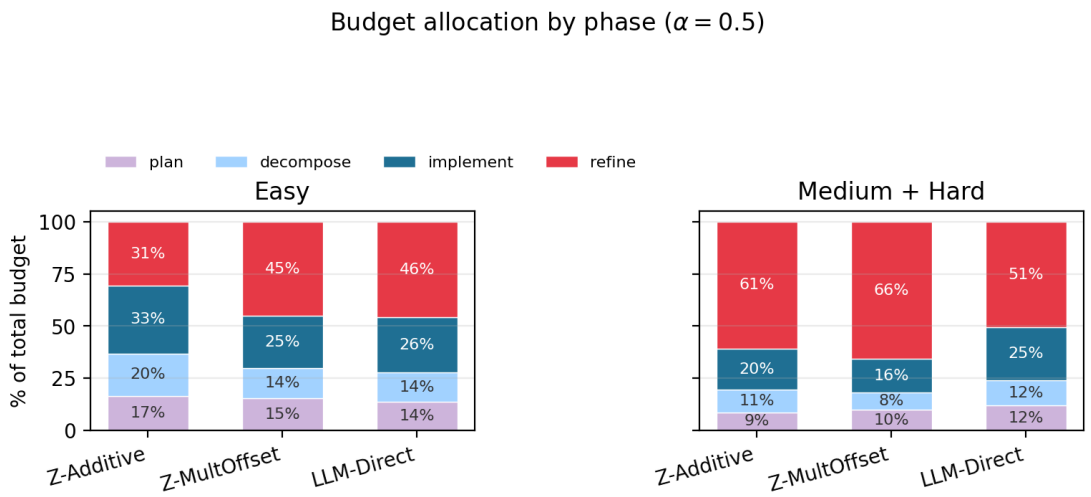

Figure 6. **Allocation distributions at $\alpha = 0.5$.** Stacked bars show the mean fraction of the total budget spent on each phase, per strategy, on easy (left) versus medium+hard (right) tasks. ZEBRA shifts spend from `implement` (easy) to `refine` (medium+hard); LLM-direct uses a near-identical split in both regimes.

10.5% vs. 10.3%; `implement` 20.7% vs. 19.1%; `refine` 56.6% vs. 58.8%). Reusing a_i as the weight, `prop_offset` does not pull the allocation meaningfully away from the symmetric `mult_offset` solution.

Why the per-phase weights do not help. A few plausible reasons, most of which point to the operationalization of the weights rather than to the dependency-modeling idea itself:

- *The weights are redundant to existing information.* Setting $w_i = a_i$ reuses the same per-phase quality ceiling that `mult_offset` already incorporates through the offset curve $1 - a_i e^{-b_i x_i}$. The additional a_i exponent reweights the product but adds no new signal, so allocations end up tracking the symmetric $w_i \equiv 1$ solution.
- *On hard tasks, every phase matters.* In the regime where `mult_offset` most clearly dominates `additive` (hard tier at $\alpha = 0.5$), tilting away from balance costs more than it saves – and we observe `prop_offset` losing to `mult_offset` on hard tasks at $\alpha = 0.5$ (0.1175 vs. 0.1300).

We expect dependency-weighting could help in settings where the weight carries information beyond a_i e.g., explicit per-phase weights elicited from a separate prompt, or learned from observed cascades but reusing a_i as the weight does not move the allocation meaningfully in the 4-phase coding pipeline we evaluate.

α	Tier (n)	add.	prop.	mult.	llm
0.3	Easy (50)	**38**	5	3	4
0.5	Easy (50)	**32**	7	2	9
	Medium (50)	**18**	13	13	6
	Hard (50)	14	9	**21**	6
	All (150)	**64**	29	36	21
0.8	Easy (50)	**33**	4	7	6
	Medium (50)	**18**	16	10	6
	Hard (50)	**21**	9	13	7
	All (150)	**72**	29	30	19

Table 10. **Per-task win counts.** Number of tasks (out of the indicated n) on which each strategy achieves the highest per-task mean score among the four reported strategies. `add.` = `zebra-additive`, `prop.` = `zebra-prop_offset`, `mult.` = `zebra-mult_offset`. The $\alpha = 0.3$ row is restricted to the easy tasks (Section 5.4); $\alpha \in \{0.5, 0.8\}$ rows are from the full 150-task run. `zebra-additive` is the per-task win-count leader at every aggregate budget, and `zebra-mult_offset` takes over at the hard tier under $\alpha = 0.5$. LLM-direct wins only 13–14% of tasks at $\alpha \geq 0.5$, falling to 8% at $\alpha = 0.3$. Bold marks the leader per row.

α	Tier (n)	prop mean	mult mean	Δ (prop−mult)	p vs mult
0.5	Easy (50)	0.9347	0.9385	−0.0039	0.28 ^{ns}
	Medium (50)	0.5473	0.5506	−0.0033	0.91 ^{ns}
	Hard (50)	0.1175	0.1300	−0.0125	0.10 ^{ns}
	Med+Hard (100)	0.3324	0.3403	−0.0079	0.33 ^{ns}
	All (150)	0.5331	0.5397	−0.0066	0.15 ^{ns}
0.8	Easy (50)	0.9563	0.9582	−0.0019	0.59 ^{ns}
	Medium (50)	0.5945	0.5772	+0.0173	0.083 ^{ns}
	Hard (50)	0.1365	0.1376	−0.0011	0.82 ^{ns}
	Med+Hard (100)	0.3655	0.3574	+0.0081	0.22 ^{ns}
	All (150)	0.5624	0.5576	+0.0048	0.40 ^{ns}

Table 11. **`prop_offset` vs. `mult_offset`, head-to-head on 150 tasks (15 runs per cell).** Mean per-task score, the difference Δ = prop − mult, and the two-sided paired Wilcoxon signed-rank p-value on per-task mean score against `mult_offset`. The displayed p is raw; significance markers reflect Benjamini–Hochberg adjustment over the 10 tests in this family ($^{*}p_{\text{BH}} < 0.05$, $^{**}p_{\text{BH}} < 0.01$, $^{***}p_{\text{BH}} < 0.001$, ns = not significant). **No cell is significant at $p_{\text{raw}} < 0.05$, and therefore none is significant under BH.** The sign of Δ flips between budget regimes ($\Delta < 0$ at $\alpha = 0.5$ overall, $\Delta > 0$ at $\alpha = 0.8$ overall). Win counts on the all-tasks rows are prop 50 / mult 62 / ties 38 at $\alpha = 0.5$ and prop 57 / mult 57 / ties 36 at $\alpha = 0.8$. Conclusion: the LLM-elicited α_i weights do not move the per-task score reliably away from the symmetric `mult_offset` ($w_i \equiv 1$) baseline.

D.13. Does this transfer? HumanEval and CodeContests

To check whether ZEBRA's APPS gains generalize to other coding benchmarks we run two transfer datasets, each at the tight budget $\alpha = 0.5$ only:

- **HumanEval** (Chen et al., 2021). We extract 50 candidate tasks via `extract_humaneval_tasks.py`, screen them with 15 no-budget runs as in Appendix D.2, and retain 29 tasks (after dropping trivially-solved cases and tasks without sufficient cost variance). The retained pool is dominated by the easy tier (28 easy, 1 medium, 0 hard) – a regime where the APPS main results predict that ZEBRA's advantage should be small.
- **CodeContests** (Li et al., 2022). We extract 50 candidate tasks via `extract_codecontests_tasks.py`, screen identically, and retain 23 tasks. The retained pool is balanced toward the harder end (8 easy, 5 medium, 10 hard) and is the closest analogue to APPS' tier mix among the transfer datasets we tested.

Both runs use the same pipeline, models and budget definition as the APPS experiments; only the task pool changes.

HumanEval ($n = 29$, $\alpha = 0.5$). Both ZEBRA variants finish above LLM-direct on per-task mean and on NB-retention; `zebra-mult_offset` reaches significance at the raw level ($p = 0.025$). The headline gaps are small in absolute terms

Strategy	Mean	NB ret.	vs LLM (W/L/T)	p_{raw}
`zebra-additive`	0.9626	98.6%	9/2/18	0.0754 ns
`zebra-mult_offset`	**0.9672**	**99.1%**	**11**/1/17	0.0253*
`llm_cot`	0.9463	97.0%	8/6/15	0.8752 ns
`llm`	0.9419	96.5%	—	—

Table 12. **HumanEval transfer at $\alpha = 0.5$ ($n = 29$ tasks, 15 runs per cell).** Per-task mean F1 score, NB retention, and paired Wilcoxon signed-rank test vs. LLM-direct. `zebra-mult_offset` reaches significance at the raw level; `zebra-additive` is in the same direction but underpowered. 28 of 29 retained tasks fall in the easy tier, where the APPS main results predict the smallest advantage.

(+2–3 points) because 28 of 29 retained tasks are easy and the budget is rarely binding – exactly the regime where the APPS main results show the smallest advantage (Table 2, easy column). LLM-CoT does not close the gap: it scores $+0.004$ over plain LLM-direct, n.s.

Strategy	Mean	NB ret.	vs LLM (W/L/T)	p_{raw}
`zebra-additive`	**0.3824**	**80.3%**	**12**/7/4	0.1712 ns
`zebra-mult_offset`	0.3681	77.3%	10/8/5	0.3958 ns
`llm`	0.3534	74.2%	—	—

Table 13. **CodeContests transfer at $\alpha = 0.5$ ($n = 23$ tasks, 15 runs per cell).** Per-task mean F1 score, NB retention, and paired Wilcoxon signed-rank test vs. LLM-direct. Both ZEBRA variants finish above LLM-direct on mean and NB-retention with per-task wins favouring ZEBRA, but neither cell reaches significance at this sample size; we treat CodeContests as an underpowered generalization probe rather than a primary result.

CodeContests ($n = 23$, $\alpha = 0.5$). Both ZEBRA variants finish above LLM-direct in mean and NB-retention, with the per-task wins favouring ZEBRA (12/7/4 for additive), but neither cell reaches significance at $n = 23$ – the screened pool is small. The signal is in the right direction but underpowered. We treat CodeContests as a generalization probe rather than as a primary result.

What transfers and what does not. The patterns from the APPS main results carry over: (i) ZEBRA's advantage is largest where budget is binding and the task is non-trivial – on HumanEval, where $28/29$ tasks are easy and budget is loose, the gap is small but in the right direction; (ii) curve-aware allocation does not hurt in the easy regime even with mult's product objective, contradicting the concern that the multiplicative formulation might over-fund refine on trivially-solved tasks; (iii) chain-of-thought prompting on the LLM-direct controller does not recover the gap on either dataset, just as on APPS. We do not claim CodeContests is statistically distinguishable from LLM-direct; we report it because the screened pool was tier-mixed and the direction of the effect is consistent with the APPS results.

D.14. Sensitivity to curve-estimation noise

We isolate the effect of curve accuracy by bypassing the live controller and injecting curves with controlled levels of synthetic noise. For each task we average the controller's (a_i, b_i) estimates across the 15 runs of the main benchmark (`zebra-additive`, $\alpha = 0.5$); these averaged curves are treated as a stable per-task "ground-truth." We then perturb each parameter with multiplicative Gaussian noise $a_i' = a_i(1 + \mathcal{N}(0, \sigma))$, $b_i' = b_i(1 + \mathcal{N}(0, \sigma))$, with values clipped to $a \in [0.01, 1]$ and $b \in [0.01, \infty)$. We test $\sigma \in \{0\%, 10\%, 50\%\}$. The pipeline, models, and budget ($\alpha = 0.5$) are held fixed; only the injected curves change. Each (task, σ) cell is repeated 5 times, yielding $150 \times 3 \times 5 = 2{,}250$ runs.

Table 14 reports per-task mean scores and paired Wilcoxon tests against the live controller and against the LLM-direct and Uniform baselines. **The allocation is robust to estimation error.** At $\sigma = 50\%$, where parameters can be halved or doubled (e.g., a true $a = 0.6$ becomes anywhere in roughly $[0.3, 0.9]$), mean score drops only from 0.5305 to 0.5165 – a 1.4-point gap that is not significant against the live controller ($p = 0.08$) but remains significant against Uniform ($p = 0.03$). At $\sigma \leq 10\%$, all baseline comparisons are highly significant ($p < 10^{-3}$). The shape (diminishing returns) of the curves matters more than the precise parameter values: the optimizer maps roughly the same (a, b) region to roughly the same allocation, so the curves only need to be *directionally* correct.

The $\sigma = 0\%$ averaged-curve condition slightly outperforms the live controller (0.5305 vs. 0.5258), so per-run estimation *variance* can be a source of allocation suboptimality.

Strategy	Mean	vs. live (p)	vs. LLM (p)	vs. Uniform (p)
Live controller (additive)	0.5258	—	0.0013^{**}	$< 10^{-4***}$
$\sigma = 0\%$ (averaged)	0.5305	0.52^{ns}	0.0009^{***}	$< 10^{-4***}$
$\sigma = 10\%$	0.5384	0.32^{ns}	0.0004^{***}	$< 10^{-4***}$
$\sigma = 50\%$	0.5165	0.08^{ns}	0.30^{ns}	0.033^{*}
LLM-direct	0.5038	0.0013^{**}	—	0.025^{*}
Uniform	0.4883	$< 10^{-4***}$	0.025^{*}	—

Table 14. **Sensitivity to curve-estimation noise** ($n = 150$ tasks, $\alpha = 0.5$). Multiplicative Gaussian noise on (a, b) with $\sigma \in \{0\%, 10\%, 50\%\}$. p-values are paired Wilcoxon signed-rank, two-sided. Even at $\sigma = 50\%$, the allocation does not significantly underperform the live controller.

D.15. Controller-model variance: gpt-4o vs. gpt-4o-mini

We re-run only the allocate phase in the $\alpha = 0.5$ benchmark with `gpt-4o-mini` replacing `gpt-4o` in the curve-estimation step and compare the resulting per-task mean fractional allocations on `additive` and `mult_offset`. The two controllers produce highly correlated allocations (Table 15). On `additive` the per-phase-averaged Pearson correlation is 0.948 but the mean absolute fractional difference is 5.4 pp, driven by the larger-model controller pushing $\sim$15 pp from `refine` back to `decompose/implement` on easy tasks. On `mult_offset` the controllers agree more tightly on the actual fractions (MAE 1.75 pp) with a similar overall correlation (0.924). Within either controller, run-to-run standard deviation is small (≤ 4.3 pp on `refine`, ≤ 1.4 pp on the upstream phases). The framework's allocation is therefore not pinned to a particular controller LLM.

Objective	Tier	Pearson	Spearman	MAE	P90
`additive`	all	0.948	0.863	0.054	0.134
	easy	0.848	0.644	0.083	0.194
	medium	0.961	0.869	0.046	0.114
	hard	0.946	0.555	0.034	0.068
`mult_offset`	all	0.924	0.910	0.018	0.048
	easy	0.789	0.642	0.030	0.088
	medium	0.935	0.890	0.016	0.045
	hard	0.964	0.791	0.007	0.017

Table 15. **Controller-model agreement on per-task fractional allocations** (`gpt-4o` vs. `gpt-4o-mini`, $n = 150$, $\alpha = 0.5$). Pearson/Spearman correlations are computed *independently per phase* on the per-task mean fractional allocations, then averaged across the four phases; **MAE** is the mean absolute disagreement on a phase fraction across all $4 \times n$ phase-task pairs; **P90** is the 90th percentile of the same disagreements (in pp of budget). The two controllers agree closely on `mult_offset` and reasonably on `additive`; the largest disagreements concentrate on easy tasks, where any allocation works.

D.16. Per-task adaptation: comparison to a fixed average split

To test whether ZEBRA's gain comes from *per-task adaptation* or merely from knowing the right average ratio, we construct a **Fixed-Avg** baseline that pre-computes the global mean per-phase share of `zebra-additive` across all $150 \times 15 = 2{,}250$ samples at $\alpha = 0.5$ ($11.3/14.0/24.1/50.6\%$ for plan/decompose/implement/refine) and applies that fixed split to every task. Fixed-Avg has *zero* controller cost (the ratios are pre-computed) and is otherwise identical to ZEBRA in pipeline, models, and per-task total budget; it is the strongest non-adaptive baseline available without training.

Takeaway. On easy tasks, dynamic ZEBRA and Fixed-Avg are statistically indistinguishable – when the budget is loose, the choice of split does not matter. On medium and hard tasks (the regimes Section 5.3 identifies as where allocation matters), both ZEBRA variants outperform Fixed-Avg in absolute retention by $+9$ to $+22$ pp NB-retention. The gap is statistically significant in 3 of the 4 medium/hard cells ($p < 0.01$); the one exception is `additive` vs. Fixed-Avg on hard ($p = 0.17$), where `mult_offset` retains significance ($p = 0.009$). Even so, Fixed-Avg uses ZEBRA's own *average* split: the benefit ZEBRA delivers is not the average ratio (any practitioner could hard-code that), but the per-task variation around it.

	all	easy	medium	hard
Fixed-Avg (mean score)	0.5048	0.9253	0.4866	0.1023
Fixed-Avg (NB retention)	83.4%	95.4%	78.7%	74.8%
`zebra-additive` (mean)	0.5258	0.9216	0.5324	0.1234
`zebra-additive` (NB ret.)	89.4%	95.1%	87.7%	84.7%
p vs. Fixed-Avg	0.011^{*}	0.48^{ns}	0.008^{**}	0.17^{ns}
`mult_offset` (mean)	0.5397	0.9385	0.5506	0.1300
`mult_offset` (NB ret.)	94.3%	97.0%	90.1%	96.3%
p vs. Fixed-Avg	$< 10^{-4***}$	0.06^{ns}	$< 10^{-3***}$	0.009^{**}

Table 16. **Dynamic ZEBRA vs. Fixed-Avg.** NB retention is computed as a mean of per-task ratios, $\text{mean}_i(s_i/s_i^{\text{NB}})$, averaged over tasks with $s_i^{\text{NB}} \geq 0.01$. Both ZEBRA variants are statistically indistinguishable from Fixed-Avg on easy tasks – where any reasonable split works – and outperform it on medium and hard tasks. The gap is statistically significant in 3 of 4 medium/hard cells: on medium tasks for both variants (`additive` $p = 0.008$, `mult_offset` $p < 10^{-3}$), and on hard tasks for `mult_offset` ($p = 0.009$) but not `additive` ($p = 0.17$). The gain from ZEBRA therefore comes from per-task adaptation, not from the choice of average ratio.

D.17. Controller overhead vs. allocation gain

We answer two questions: (a) *how large is the controller overhead in practice?* and (b) *is the allocation gain large enough to justify it, even against a comparator with substantially more budget?*

(a) Controller overhead statistics. The controller's curve-estimation call has an essentially fixed cost per task (it depends on the task description, not on the budget), so its share of the total spend grows as the budget shrinks. Table 17 reports overhead summaries per strategy at the main benchmark budget ($\alpha = 0.5$).

Strategy	n runs	Ctrl. \$ (mean)	Total \$	Phase \$	Ctrl. / total spent
`zebra-additive`	735	\$0.00362	\$0.01244	\$0.00882	29.1%
`zebra-mult_offset`	735	\$0.00362	\$0.01242	\$0.00880	29.2%
LLM-direct	735	\$0.00234	\$0.00991	\$0.00757	23.6%

Table 17. **Controller overhead statistics.** Per-strategy summaries at the main benchmark budget ($\alpha = 0.5$), restricted to the 49 APPS tasks with valid raw phase-spending data ($49 \times 15 = 735$ runs each); the remaining 101 tasks are excluded because of a logging artifact. ZEBRA's controller overhead is 29.1% of total spend for `additive` and 29.2% for `mult_offset` (33.1% of the budget in both cases).

At $\alpha = 0.5$ the controller consumes 29.1% of `zebra-additive` total spend and 29.2% of `mult_offset` (about 33% of the per-task budget for both). Because the controller call has a roughly fixed cost per task while the phase budget shrinks with α, this share grows at tighter budgets and becomes a non-trivial fraction of the total.

(b) Does the allocation gain justify the overhead? We compare ZEBRA at $\alpha = 0.5$ against *Uniform* at $\alpha = 0.8$, a comparator with 60% *more* total budget and zero controller overhead. If allocation alone is doing real work, ZEBRA should still win even at this budget disadvantage. Table 18 reports the head-to-head.

Comparison	Mean diff.	95% CI	Wilcoxon p	W/L/T
`zebra-additive`@α=0.5 vs. Uniform@α=0.8	+0.0273	[+0.013, +0.042]	1.6×10^{-4}	59/33/58
`mult_offset`@α=0.5 vs. Uniform@α=0.8	+0.0412	[+0.025, +0.057]	$<10^{-6}$	69/20/61
`zebra-additive`@α=0.5 vs. Uniform@α=0.5	+0.0375	[+0.023, +0.052]	3.0×10^{-6}	—
`mult_offset`@α=0.5 vs. Uniform@α=0.5	+0.0514	[+0.035, +0.068]	$<10^{-6}$	—

Table 18. **Allocation gain vs. controller overhead.** Top: ZEBRA at $\alpha = 0.5$ vs. Uniform at $\alpha = 0.8$ – a comparator given 60% more total budget (1.6×) but zero controller cost. Both ZEBRA variants still win significantly. Bottom: same-budget reference ($\alpha = 0.5$ on both sides). All comparisons are paired Wilcoxon signed-rank on $n = 150$ APPS tasks (15 runs per cell). W/L/T sums to 150 for the top block.

Both ZEBRA variants beat Uniform-0.8 significantly, despite the 60% budget disadvantage. ZEBRA's controller overhead at $\alpha = 0.5$ – ∼33% of budget for both `additive` and `mult_offset` (∼29% of total spend) – is therefore more than compensated by the allocation gain.

D.18. Empirical validation of pipeline monotonicity

ZEBRA's solver assumes each phase is non-decreasing in spend (Section 3.1). Recent work on "overthinking" (Chen et al., 2026; Sui et al., 2025) shows this assumption can fail for individual chain-of-thought prompts: very long reasoning traces can hurt accuracy on simple tasks. We therefore empirically validate the assumption *for our pipeline* by fitting a within-task fixed-effects regression of final score on total realized spend, separately for each tier:

$$\text{score}_{t,r} = \mu_t + \beta\,\text{spend}_{t,r} + \varepsilon_{t,r},$$

where t indexes task, r indexes run, and μ_t is a per-task intercept. Pooling 50 tasks per tier ($n = 6{,}000$ per tier; 18,000 total runs across our APPS benchmark and ablations) gives the within-task slopes in Table 19.

Tier	n runs	$\hat{\beta}$ (score per \$)	95% CI	p
Easy	6,000	9.37	$[6.56, 12.19]$	$< 10^{-6}$
Medium	6,000	5.19	$[4.15, 6.24]$	$< 10^{-6}$
Hard	6,000	1.83	$[1.19, 2.46]$	$< 10^{-6}$

Table 19. **Pipeline is monotone in total spend on every tier.** Within-task fixed-effects slopes of score on total realized spend, fit on 18,000 runs (150 tasks $\times$120 runs). $\hat{\beta} > 0$ at $p < 10^{-6}$ in every tier validates the non-decreasing-utility assumption underlying the solver.

The slope is positive and highly significant in every tier, confirming that *within* a task, more budget yields higher score. The slope shrinks with difficulty – $9.4 \to 5.2 \to 1.8$ score-per-dollar from easy to hard – consistent with the diminishing-returns shape that motivates ZEBRA's saturating exponential. The assumption is therefore appropriate for this pipeline; the overthinking failure mode reported by Chen et al. (2026); Sui et al. (2025) would manifest as a *negative* within-task slope on at least one tier, which we do not observe.

E. HotpotQA Generality Experiment

This appendix details the HotpotQA generality experiment summarized in Section 5.9. The goals are to (i) move ZEBRA out of the coding domain, (ii) use a different number of phases, and (iii) use a different orchestration framework – so that any positive transfer cannot be attributed to a coding- or LangGraph-specific artefact.

E.1. Dataset and task selection

We use the HotpotQA distractor validation split as a source of multi-hop questions with short gold answers and difficulty labels; the pipeline answers from the model's parametric knowledge and does not consume the distractor paragraphs. We sample 200 questions deterministically with seed $= 42$ via `extract_hotpotqa_tasks.py`. Each task is stored as a JSON file with fields {`question, answer, type, level, supporting_facts`}. Only question and answer are used.

Screening. All 200 tasks are screened with the no-budget pipeline (15 runs each, `gpt-4o-mini`, max-workers $= 8$). Per task we record the mean, median, and standard deviation of cost (USD) and of F1 score across the 15 runs.

Filtering. Tasks are retained if they satisfy both:

- Non-trivial score range: $0.20 \le \overline{\text{F1}}_{\text{NB}} \le 0.85$. Below 0.2 the model essentially cannot answer regardless of budget; above 0.85 the model essentially always answers correctly, leaving no headroom for allocation to matter.
- Cost stability: $\text{CV}_{\text{cost}} < 0.35$ (where $\text{CV} = \text{stdev}/\text{median}$). This ensures the per-task budget reference $\bar{c}$ is meaningful.

48 of 200 tasks pass both filters. Splitting at the median no-budget F1 (0.5) gives 25 *easy* ($\overline{\text{F1}} \in [0.5, 0.85]$) and 23 *hard* ($\overline{\text{F1}} \in [0.2, 0.5)$) tasks.

Tier	Strategy	Mean F1	Cost ($)	NB retention	p vs LLM	p vs Uniform
Overall ($n = 48$)	zebra-additive	**0.473**	0.000281	**92.1**%	$1\times10^{-4}{}^{***}$	0.65^{ns}
	zebra-mult_offset	0.463	0.000280	90.0%	0.0007^{***}	0.51^{ns}
	uniform	0.469	0.000265	91.3%	0.0004^{***}	—
	llm	0.400	0.000181	77.8%	—	0.0004^{***}
Easy ($\bar{s} \geq 0.5$, $n = 25$)	zebra-additive	0.589	0.000270	86.8%	0.011^{*}	0.86^{ns}
	zebra-mult_offset	0.590	0.000271	86.9%	—	0.75^{ns}
	uniform	**0.601**	0.000255	**88.5**%	0.023^{*}	—
	llm	0.526	0.000160	77.4%	—	0.023^{*}
Hard ($\bar{s} < 0.5$, $n = 23$)	zebra-additive	**0.347**	0.000292	**97.8**%	0.0016^{**}	0.44^{ns}
	zebra-mult_offset	0.331	0.000291	93.3%	—	0.62^{ns}
	uniform	0.335	0.000277	94.4%	0.006^{**}	—
	llm	0.272	0.000204	78.2%	—	0.006^{**}

Table 20. **HotpotQA generality experiment** ($\alpha = 0.5$, 15 runs per cell, `gpt-4o-mini`). NB retention is computed as a mean of per-task ratios, $\text{mean}_i(\text{F1}_i/\text{F1}_i^{\text{NB}})$, averaged over tasks with $\text{F1}_i^{\text{NB}} \geq 0.01$. Reference no-budget means: $\overline{\text{F1}}_{\text{overall}} = 0.514$, $\overline{\text{F1}}_{\text{easy}} = 0.679$, $\overline{\text{F1}}_{\text{hard}} = 0.334$. Tasks are split at the median no-budget F1. **Both ZEBRA variants significantly beat LLM-direct** ($p < 10^{-3}$ overall) and **tie Uniform** ($p > 0.4$): the optimal split on this benchmark is close to balanced and ZEBRA recovers that, while LLM-direct does not (cf. Table 21). Significance markers: $^{*}p < 0.05$, $^{**}p < 0.01$, $^{***}p < 0.001$, ns = not significant.

E.2. Pipeline

The HotpotQA pipeline is intentionally different from the APPS pipeline. It uses *three* phases in plain Python (no LangGraph), chained as `direct` → `research` → `synthesize`:

direct One-shot answer to the multi-hop question. Produces a candidate answer.

research Decomposes the question into 2–4 sub-questions and answers each with a specific fact. Produces *evidence only* – not a final answer.

synthesize Combines the `direct` answer with the `research` evidence; explicitly instructed to keep the `direct` answer unless the evidence contradicts it. Produces a final candidate answer.

Each phase supports iterative refinement within its allocated budget (initial generation; subsequent iterations may refine or output `NO_CHANGE`; the phase terminates on budget exhaustion, two consecutive no-changes, or a maximum-iteration cap). Budget enforcement uses the same `BudgetEnforcedLLM` interface as APPS. The pipeline keeps the highest-F1 candidate across phases, ensuring monotonicity in number of phases.

E.3. Strategies

We compare four strategies, all sharing the same pipeline and controller model (`gpt-4o-mini`):

- **zebra-additive** – ZEBRA's controller estimates (a_i, b_i) for each of the three QA phases; allocations are solved with the additive knapsack of Section 4.2.1 (`solver.py` unchanged from APPS).
- **zebra-mult_offset** – same controller, the multiplicative-offset solver of Section 4.2.2 (`solver.py` unchanged).
- **llm** – LLM-direct: the controller, given the same pipeline description and total budget, outputs per-phase USD allocations as JSON; the allocations are scaled to the total budget.
- **uniform** – equal split, $B/3$ per phase.

A no-budget run (unconstrained pipeline) provides the oracle ceiling and the per-task budget reference.

E.4. Budget protocol and metrics

Per task we set $B = \alpha \cdot \tilde{c}_{\text{NB}}$ with $\alpha = 0.5$ and $\tilde{c}_{\text{NB}}$ the per-task median no-budget cost. Each (task, strategy) cell is run for 15 independent runs.

We use the standard HotpotQA (Yang et al., 2018) / SQuAD (Rajpurkar et al., 2016) evaluation: predictions and gold answers are normalized (lowercase, strip articles, strip punctuation, collapse whitespace) and scored by token-level F1 between the resulting tokens. Token F1 is the primary metric (more granular than exact match for short factoid answers).

E.5. Results

The headline results are reported in Table 20 (overall + per tier) and in Table 21 (per-phase allocations).

Tier	Strategy	direct	research	synthesize	Avg. spent ($)
Overall ($n = 48$)	zebra-additive	30.3%	39.4%	30.4%	0.000281
	zebra-mult_offset	29.4%	42.6%	28.0%	0.000280
	uniform	33.3%	33.3%	33.3%	0.000265
	llm	68.0%	19.8%	12.2%	0.000181
Easy ($n = 25$)	zebra-additive	31.2%	39.0%	29.8%	0.000270
	zebra-mult_offset	30.0%	42.2%	27.7%	0.000271
	uniform	33.3%	33.3%	33.3%	0.000255
	llm	71.2%	16.4%	12.5%	0.000160
Hard ($n = 23$)	zebra-additive	29.3%	39.8%	31.0%	0.000292
	zebra-mult_offset	28.7%	42.9%	28.4%	0.000291
	uniform	33.3%	33.3%	33.3%	0.000277
	llm	64.7%	23.5%	11.8%	0.000204

Table 21. **HotpotQA per-phase budget shares** (%) and realized spend, averaged across runs and tasks. *Both ZEBRA variants converge to a near-balanced allocation* ($\sim 30/40/30$), within a few points of Uniform.

ZEBRA significantly beats LLM-direct. Overall, `zebra-additive` retains 92.1% of the no-budget F1 versus 77.8% for LLM-direct – a $+14.3$pp gap on NB retention (equivalently $+0.073$ on absolute mean F1: 0.473 vs. 0.400; $p = 1{\times}10^{-4}$ paired Wilcoxon on per-task mean F1, $n = 48$). The gap holds in both tiers: $+9.4$pp NB retention on easy tasks ($p = 0.011$), and widens to $+19.6$pp NB retention on the hard tier ($p = 0.0016$). Uniform also beats LLM-direct ($+13.5$pp NB retention overall, $p = 0.0004$), so the issue is squarely with the LLM controller's allocation, not with the pipeline itself.

ZEBRA and Uniform are statistically tied. All differences below are NB-retention gaps in percentage points (Wilcoxon tests are computed on per-task mean F1). `zebra-additive` vs. Uniform: $+0.8$pp overall ($p = 0.65$), -1.7pp easy ($p = 0.86$), $+3.4$pp hard ($p = 0.44$). `zebra-mult_offset` vs. Uniform: -1.3pp overall ($p = 0.51$), and similarly non-significant per tier. None of the comparisons reach significance at $\alpha = 0.05$. For reference, the corresponding absolute-F1 differences are small: `zebra-additive` vs. Uniform is $+0.004$ overall, -0.012 easy, $+0.012$ hard.

Why Uniform is so competitive here. The LLM-direct allocation is sharply skewed – $\sim 68\%$ on `direct`, $\sim 20\%$ on `research`, $\sim 12\%$ on `synthesize` – and is consistently dominated. The successful strategies all land near a balanced split: ZEBRA-additive averages $30/39/30$ and ZEBRA-mult_offset averages $29/43/28$, both within ~ 10pp of the $33/33/33$ uniform reference (Table 21). On these QA tasks the three phases contribute comparably to end-to-end F1, so any near-balanced split lands close to the optimum. ZEBRA does not beat Uniform because there is essentially no allocation gap to exploit; the gap to LLM-direct, on the other hand, is large because LLM-direct is qualitatively miscalibrated.

This is the generality result. On APPS, the same ZEBRA controllers learn a sharply skewed allocation ($\sim 11/14/24/51\%$ across plan/decompose/implement/refine, Table 9); on HotpotQA they converge to a balanced one. The framework adapts the *shape* of the allocation to what the task structure demands. On benchmarks where balance happens to be optimal, the controller recovers near-balance and matches Uniform, while still beating an LLM that allocates by intuition.

ZEBRA's allocation is also consistent across tiers on HotpotQA, in contrast to APPS. A second axis of adaptation is within-benchmark: do per-tier allocations differ? On APPS they do – `refine` alone moves by $\sim$15–30pp between easy and medium/hard at $\alpha = 0.5$ (Sec. 5.5, Table 9). On HotpotQA they do not: `zebra-additive` allocates $31.2/39.0/29.8$ on easy versus $29.3/39.8/31.0$ on hard (per-phase shifts of 1.9, 0.8, 1.2 pp respectively); `zebra-mult_offset` behaves the same way (per-tier shifts ≤ 1.3 pp on every phase). The per-task standard deviation of each allocation share is also small

(3–5 pp), so the tier-flat average is not hiding bimodal behavior. The reading we draw is: *ZEBRA's per-tier adaptation is itself task-dependent*. On APPS the difficulty signal moves the controller's curves and the optimal allocation correspondingly shifts; on HotpotQA the difficulty signal does not change which phase is the bottleneck, the controller's curves move proportionally across tiers, and the allocation stays put. "Adapt when adaptation helps; do not adapt when it does not" is a property of the controller.

Why hard-tier retention is higher than easy-tier retention. The hard tier preserves $\sim 98\%$ of NB F1 versus $\sim 87\%$ on the easy tier (Table 20). This is not a contradiction: hard tasks have low absolute F1 even unconstrained ($\overline{\text{F1}}_{\text{NB}} = 0.334$), so halving the budget moves the mean by only ~ 0.006 F1 – the model's knowledge is the bottleneck, not compute. Easy tasks ($\overline{\text{F1}}_{\text{NB}} = 0.679$) are within reach at unconstrained budget but at HotpotQA's micro-budgets ($\bar{c}_{\text{NB}} = \$3.3{\times}10^{-4}$) the $\alpha = 0.5$ cut directly bites tasks the model can otherwise solve. The interaction between budget pressure and absolute headroom is the same one we observe on APPS (Sec. 5.4); it just expresses itself differently when the absolute scores are far below saturation.

E.6. What does *not* carry over from APPS

Two parts of the APPS-side story do not reproduce here:

- *ZEBRA does not beat Uniform on this benchmark.* Uniform is the right answer here; ZEBRA *is* Uniform, just discovered zero-shot. We frame this as a feature (the framework adapts) rather than a strict win, and we state it plainly in the body (Section 5.9).
- *The mult_offset variant does not lift the hard-tier mean above additive.* On APPS, `mult_offset` is the strongest variant on hard tasks because the product form rewards funding the weakest link; at HotpotQA's near-balanced optimum the two objectives produce nearly identical allocations and similar scores (0.473 vs. 0.463 overall).

The qualitative story – ZEBRA dominates LLM-direct, and the allocation it learns is task-appropriate – carries over cleanly.

F. Hybrid (Mid-Pipeline) Re-allocation

ZEBRA as presented in the main paper allocates the budget once, before execution. A natural extension is to add an *online* re-allocation step that, partway through the pipeline, re-estimates the remaining phases' utility curves conditioned on what has already been produced and re-solves the water-filling problem on the leftover budget. We implemented and evaluated this hybrid variant on the same APPS pipeline used in the main paper. The result is negative: hybrid re-allocation does not improve over one-shot ZEBRA in this setting. We document the experiment here for completeness and to inform future work in longer or more uncertain pipelines.

F.1. Hybrid pipeline

The static ZEBRA pipeline,

$$\texttt{allocate} \rightarrow \texttt{plan} \rightarrow \texttt{decompose} \rightarrow \texttt{implement} \rightarrow \texttt{refine} \rightarrow \text{END},$$

is augmented with two new nodes inserted after `decompose`:

$$\texttt{allocate} \rightarrow \texttt{plan} \rightarrow \texttt{decompose} \rightarrow \texttt{judge} \rightarrow \texttt{reallocate} \rightarrow \texttt{implement} \rightarrow \texttt{refine} \rightarrow \text{END}.$$

`judge` node. The same controller LLM (`gpt-4o`) is shown the original task and the partial output (`plan` + `decompose`) and asked to rate the completeness and correctness of the decomposition on a 1–5 scale with a one-sentence justification (returned as JSON).

`reallocate` node. Given the original task, the partial output, the already-spent budget $s = s_{\texttt{plan}} + s_{\texttt{decompose}}$, the total budget B, and (optionally) the judge score and reason, the controller re-estimates the two operating points (tokens_basic, tokens_great) for the remaining phases `implement` and `refine`. The same two-point fitting procedure (Section 4.1, full derivation in Appendix B.4) converts these into (a_i, b_i), and the same water-filling solver (Section 4) is applied to the remaining budget $B' = B - s$ over $\{\texttt{implement}, \texttt{refine}\}$, yielding new caps $(x'_{\texttt{implement}}, x'_{\texttt{refine}})$ that replace the existing caps in pipeline state.

F.2. Strategies

We compared three cells:

- **`zebra-additive` (baseline).** Static, one-shot ZEBRA with the additive aggregation, as used in the main paper.
- **`hybrid-no-judge`.** Pre-allocates exactly as static ZEBRA, then runs `reallocate` after `decompose` *without* a judge score.
- **`hybrid-with-judge`.** Same as `hybrid-no-judge`, but the `reallocate` prompt also receives the judge score and reason.

All three share the task list, the controller and base-phase LLMs, and the evaluation harness. The only differences are the inserted nodes and the resulting per-phase caps.

F.3. Setup

We used the same 150 APPS interview-level tasks (50 easy + 50 medium + 50 hard) and the budget level $\alpha = 0.5$. Each hybrid strategy was run 10 times per task; the static baseline was run 15 times per task (the existing main-paper budget). All other hyperparameters, prompts, and scoring procedures match the main paper exactly.

F.4. Results

Strategy	Mean	Median	Success %
`zebra-additive` (baseline)	0.526	0.533	52.7%
`hybrid-no-judge`	0.501	0.530	49.9%
`hybrid-with-judge`	0.525	0.516	52.2%

Table 22. Hybrid re-allocation versus one-shot ZEBRA on the 150-task APPS benchmark at $\alpha = 0.5$. No pairwise comparison reaches significance (all $p_{\text{BH}} > 0.15$).

Neither hybrid variant improves over the static baseline. `hybrid-with-judge` matches the baseline within noise; `hybrid-no-judge` is numerically slightly worse.

F.5. Why hybrid does not help here

In the APPS pipeline, `allocate` is followed by only two substantive downstream phases (`implement` and `refine`) once `plan` and `decompose` have run. With only two phases left to split a residual budget across, the curve estimates produced before execution already capture task difficulty well enough. Online re-allocation is more likely to pay off in pipelines with (i) more downstream phases over which to redistribute the residual budget, (ii) higher curve-estimation uncertainty up front, or (iii) intermediate outputs whose quality varies substantially across runs in ways the controller cannot anticipate from the task description alone. We did not observe these conditions in the APPS pipeline, and report the negative result as evidence that one-shot allocation is sufficient in this regime.

G. Examples

This appendix walks through a single APPS task end-to-end so the reader can see exactly what each strategy is shown, what each strategy outputs, and how the resulting allocation translates into a per-task quality score. We pick a short, easily-understood task on which both ZEBRA variants beat every LLM-based allocator by a wide margin, so that the mechanism behind the gain is visible without distraction. The appendix is organized as follows: App. G.1 introduces the chosen task (`apps_12`). App. G.2 reports per-strategy scores at $\alpha = 0.5$ across 15 seeds; App. G.3 reports the corresponding mean per-phase allocations; App. G.4 reproduces all four allocator prompts verbatim; App. G.6 traces what happens during execution seed-by-seed and isolates the mechanism that distinguishes ZEBRA from the LLM-based allocators.

G.1. The task: `apps_12`

`apps_12` is a Codeforces-style problem from the APPS interview set. The generator must implement `solve(stdin_input: str) -> str` for the following specification:

> Vova has n trophies in a row, each either golden (G) or silver (S). He may perform *at most one swap* of any two trophies. Output the maximum length of a contiguous G-segment achievable after that swap. ($2 \leq n \leq 10^5$.)

We run all strategies at the main pressure regime $\alpha = 0.5$, which gives a per-task budget of $B \approx \$0.01795$ (this task has an unconstrained mean cost of $\bar{c} \approx \$0.0359$). Each cell is repeated for $n = 15$ seeds. Base phases use `gpt-4o-mini-2024-07-18`; refine and the controller use `gpt-4o-2024-08-06`, which is $\sim$16.7$\times$ more expensive per call than the base model. The task has 182 tests.

G.2. Per-strategy outcome

Strategy	Mean score	Success/15	Mean phase spend	vs. LLM-Direct
LLM-Direct	0.327	3/15	\$0.01339	—
LLM-CoT	0.296	3/15	\$0.01051	−0.031
ZEBRA-LLM (ablation)	0.314	2/15	\$0.01120	−0.013
Uniform (reference)	0.336	4/15	\$0.00884	+0.009
ZEBRA-additive	**0.410**	**6/15**	\$0.01442	**+0.083**
ZEBRA-mult_offset	**0.467**	**9/15**	\$0.01477	**+0.140**

Table 23. Per-task summary on `apps_12` at $\alpha = 0.5$ over 15 seeds. The per-task budget is $B \approx \$0.01795$; the "Mean phase spend" column reports realised phase-only spend, which is bounded by B (the controller's own `gpt-4o` allocation call is billed separately).

G.3. Mean allocations across 15 seeds

The six allocators place very different bets given the same budget $B \approx \$0.01795$. Table 24 reports the mean ($\pm$ std) per-phase allocation across the 15 seeds, so the pattern reflects the strategy's typical behaviour rather than any one realisation:

	Allocated ($\times 10^{-3}$)				**Realized** ($\times 10^{-3}$)	
Strategy	plan	decompose	implement	refine	Phase spend	Score
Uniform	$4.49_{\pm 0.00}$	$4.49_{\pm 0.00}$	$4.49_{\pm 0.00}$	$4.49_{\pm 0.00}$	$8.84_{\pm 0.39}$	0.336
LLM-Direct	$2.00_{\pm 0.00}$	$2.00_{\pm 0.00}$	$4.87_{\pm 0.62}$	$9.08_{\pm 0.62}$	$13.39_{\pm 0.95}$	0.327
LLM-CoT	$1.93_{\pm 0.25}$	$2.60_{\pm 0.61}$	$6.73_{\pm 1.61}$	$6.68_{\pm 1.91}$	$10.51_{\pm 3.32}$	0.296
ZEBRA-LLM (ablation)	$2.73_{\pm 0.68}$	$2.37_{\pm 0.46}$	$5.57_{\pm 0.96}$	$7.28_{\pm 1.89}$	$11.20_{\pm 2.80}$	0.314
ZEBRA-additive	$1.09_{\pm 0.09}$	$1.36_{\pm 0.09}$	$2.47_{\pm 0.12}$	$\mathbf{13.03}_{\pm 0.24}$	$14.42_{\pm 1.50}$	0.410
ZEBRA-mult_offset	$1.07_{\pm 0.09}$	$1.34_{\pm 0.08}$	$2.40_{\pm 0.20}$	$\mathbf{13.14}_{\pm 0.24}$	$\mathbf{14.77}_{\pm 1.40}$	**0.467**

Table 24. Mean per-phase allocation and phase-only realised spend across 15 seeds, in units of 10^{-3} USD. Each row sums (in expectation) to $B = \$17.95 \times 10^{-3}$; "Phase spend" is the realised total and is bounded by B. The two ZEBRA solvers concentrate budget on `refine` (mean alloc $\$13.03 \times 10^{-3}$ for additive and $\$13.14 \times 10^{-3}$ for `mult_offset`, $\sim$73% of B), with the smallest seed-to-seed standard deviation of any strategy (refine ± 0.24). LLM-Direct allocates $\$9.08 \times 10^{-3}$, LLM-CoT $\$6.68 \times 10^{-3}$, and the ZEBRA-LLM ablation $\$7.28 \times 10^{-3}$ – all between $\frac{1}{2}$ and $\frac{2}{3}$ of ZEBRA's refine budget. As Section G.6 shows, this gap is exactly what determines whether refine produces a useful revision or a regression.

G.4. Prompts used in the experiments

We include all four allocator prompts verbatim. They share the same task description and total budget; they differ only in what they ask for.

(a) ZEBRA controller prompt (`token_twopoint`, used by ZEBRA-additive, ZEBRA-mult_offset, and the ZEBRA-LLM ablation). The controller is asked for two-point token estimates and a per-phase criticality a_i, never for an allocation:

```
You are a budget allocation controller for a multi-agent coding system.
```

```
Task: {task_description}
Phase model: gpt-4o-mini-2024-07-18 (input=$0.15/1M tokens, output=$0.6/1M
tokens)
Total budget (USD): 0.01795
The system has 4 sequential phases. All phases can iterate (refine their output
with additional LLM calls) if budget allows.
1. plan -- understand the task and create a plan
2. decompose -- break the plan into implementable tasks
3. implement -- write the solution code
4. refine -- review/revise loop using gpt-4o-2024-08-06 (∼17× more expensive per
call than gpt-4o-mini)
Carefully consider the specific task above and its difficulty. For each phase,
estimate the following parameters for THIS particular task:
- tokens_basic (integer, 100--10000): total output tokens this phase needs to
produce basically acceptable output (∼50% of potential quality). Include all
iterations.
- tokens_great (integer, ≥ tokens_basic, up to 20000): total output tokens for
near-optimal output (∼90% of potential quality). Include all iterations.
- a (criticality, 0--1): how critical this phase is for this specific task. When
budget is tight, low-a phases are the first to be cut.
Respond with a JSON object in this exact format:
{{"plan": {{"tokens_basic": <int>, "tokens_great": <int>, "a": <0-1>}},
"decompose": {{...}}, "implement": {{...}}, "refine": {{...}}}}
Rules:
- Output ONLY the JSON. No prose, no markdown, no code fences.
- Use numeric values only.
- Think carefully about how complex this specific task is and how many tokens each
phase needs.
- tokens_basic must be ≥ 100. tokens_great must be ≥ tokens_basic.
```

(b) LLM-Direct allocator prompt. The LLM is asked for a final allocation in one shot:

```
You are a budget allocation controller for a multi-agent coding system.
Task: {task_description}
Total budget (USD): 0.017951
Phases 1-3 model: gpt-4o-mini-2024-07-18 (input=$0.15/1M tokens, output=$0.6/1M
tokens)
Refine model: gpt-4o-2024-08-06 (input=$2.5/1M tokens, output=$10.0/1M tokens) --
∼17× more expensive per call than gpt-4o-mini-2024-07-18.
The system has 4 sequential phases. All phases can iterate (refine their output
with additional LLM calls) if budget allows.
1. plan -- understand the task and create a plan
2. decompose -- break the plan into implementable tasks
3. implement -- write the solution code
4. refine -- review→revise loop that catches and fixes real bugs. Each iteration
= 2 LLM calls. Uses gpt-4o-2024-08-06 (∼17× cost per call vs
gpt-4o-mini-2024-07-18). Token-heavy.
Allocate the total budget across these 4 phases.
Output ONLY a JSON object mapping each phase to its USD allocation, with no extra
text.
```

(c) LLM-CoT allocator prompt. Identical to LLM-Direct except for the final sentence, which invites the controller to reason before emitting the JSON. The controller is given `max_tokens` = 800 instead of 300:

```
You are a budget allocation controller for a multi-agent coding system.
Task: {task_description}
Total budget (USD): 0.017951
Phases 1-3 model: gpt-4o-mini-2024-07-18 (input=$0.15/1M tokens, output=$0.6/1M
tokens)
Refine model: gpt-4o-2024-08-06 (input=$2.5/1M tokens, output=$10.0/1M tokens) --
∼17× more expensive per call than gpt-4o-mini-2024-07-18.
```

```
The system has 4 sequential phases [...same as above ...]
Let's think step by step about how to allocate the budget, then output a JSON
object {"plan": ..., "decompose": ..., "implement": ..., "refine": ...} with
values in USD summing to 0.017951.
```

(d) ZEBRA-LLM ablation allocator prompt. The controller call (a) is run first; its fitted utility curves are then formatted into a prompt and handed to the LLM allocator instead of to the knapsack solver:

```
You are a budget allocation controller for a multi-agent coding system.
Total budget (USD): 0.017951
Model: gpt-4o-mini-2024-07-18 (input=$0.15/1M tokens, output=$0.6/1M tokens)
Note: The refine phase uses gpt-4o-2024-08-06 which is ~17× more expensive per
call than gpt-4o-mini-2024-07-18.
The system has 4 sequential phases:
1. plan -- understand the task and create a plan
2. decompose -- break the plan into implementable tasks
3. implement -- write the solution code
4. refine -- review/revise loop to fix bugs
We have estimated the following utility curves for each phase. Each curve models
quality = a·(1 − e^(−b·budget)), where:
  - a (quality ceiling): maximum quality this phase can achieve
  - b (saturation rate): how quickly returns diminish per dollar spent
  - Higher b means the phase saturates quickly (needs less budget)
  - Higher a means the phase is more important for overall quality
Estimated curves:
  plan: quality_ceiling(a)=0.800, saturation_rate(b)=7444.31
  decompose: quality_ceiling(a)=0.700, saturation_rate(b)=4962.87
  implement: quality_ceiling(a)=0.900, saturation_rate(b)=2977.72
  refine: quality_ceiling(a)=0.600, saturation_rate(b)=1985.15
Based on these utility curves, allocate the total budget across the 4 phases to
maximize overall task quality. Phases are sequential -- upstream quality affects
downstream phases.
Output ONLY a JSON object mapping each phase to its USD allocation, with no extra
text.
```

G.5. Sample raw responses (single seed, illustrative)

The tables above report averages across 15 seeds; in this subsection we quote a single seed's raw output verbatim from the canonical seed-1 log of each strategy.

ZEBRA controller raw response. The ZEBRA-additive and ZEBRA-mult_offset strategies share the same controller call. On seed 1 the controller returns, verbatim:

```
{
  "plan": {"tokens_basic": 300, "tokens_great": 600, "a": 0.8},
  "decompose": {"tokens_basic": 400, "tokens_great": 800, "a": 0.7},
  "implement": {"tokens_basic": 800, "tokens_great": 1500, "a": 0.9},
  "refine": {"tokens_basic": 600, "tokens_great": 1200, "a": 0.6}
}
```

After fitting b_i via the geometric mean of the two-point constraints (Section 4), the per-phase curves are $(a, b)_{\text{plan}} = (0.80,\ 4963)$, $(a, b)_{\text{decompose}} = (0.70,\ 3722)$, $(a, b)_{\text{implement}} = (0.90,\ 1922)$, $(a, b)_{\text{refine}} = (0.60,\ 2481)$. Refine's b is then divided by the `gpt-4o` cost ratio 16.7, giving $b_{\text{refine}} = 148.6$. The same (a, b) vector is then handed to the additive solver and the `mult_offset` solver.

LLM-Direct raw response. The LLM-Direct allocator returns, verbatim:

```
{"plan": 0.002, "decompose": 0.002, "implement": 0.006, "refine": 0.007951}
```

A round-numbers split with $\sim$44% of B on refine.

LLM-CoT raw response. The CoT controller (`gpt-4o`, temperature 0.3) reasons for several short paragraphs and emits:

```
{"plan": 0.002, "decompose": 0.002, "implement": 0.005, "refine": 0.008951}
```

The CoT prompt rebalances slightly toward refine ($\$8.95\times10^{-3}$ vs. LLM-Direct's $\$7.95\times10^{-3}$) at the cost of implement.

ZEBRA-LLM ablation raw response. Given the curves from the controller, the LLM allocator returns:

```
{"plan": 0.003, "decompose": 0.002, "implement": 0.006, "refine": 0.006951}
```

The LLM has been told the curves (and that refine is the expensive phase) yet still under-funds refine relative to the algorithmic solver: refine alloc $\$6.95\times10^{-3}$ vs. ZEBRA-mult_offset's $\$12.92\times10^{-3}$.

ZEBRA-additive raw output. The additive solver maximises $\sum_i u_i(x_i)$ subject to $\sum_i x_i = B$. With refine's b scaled down by the `gpt-4o` cost ratio, refine has the largest marginal utility at the binding budget and absorbs the bulk of the allocation:

```
{"plan": 0.001150, "decompose": 0.001420, "implement": 0.002537, "refine":
0.012844}
```

ZEBRA-mult_offset raw output. The multiplicative-with-offset solver maximises $\prod_i(u_i(x_i)+\epsilon)$ subject to $\sum_i x_i = B$ (see Section 4.2.2); the optimum keeps every phase non-trivially funded and pushes the slowest-saturating phase (refine) to its maximum:

```
{"plan": 0.001134, "decompose": 0.001399, "implement": 0.002499, "refine":
0.012919}
```

G.6. What happens during execution

For all strategies, each phase budget is handed to a `BudgetEnforcedLLM` wrapper that runs up to 10 iterations and stops when (i) the phase output stops changing or (ii) the phase budget is exhausted. Refine specifically implements a review–revise pair (two `gpt-4o-2024-08-06` calls per round) and a candidate selection step that compares the implement output against any revisions on the 182-test internal grader; the highest-scoring candidate is shipped.

For `apps_12` at $\alpha = 0.5$, refine starts *on $\geq 14/15$ seeds for every strategy – but only the two ZEBRA solvers reliably let the revise call* finish. Across the four non-ZEBRA strategies (LLM-Direct, LLM-CoT, ZEBRA-LLM ablation, Uniform), the revise call hits the wrapper's per-call output-token budget mid-write and either logs `[revise] Output missing END_OF_OUTPUT token ...Truncated output to parseable prefix` or earlier `Budget below preferred length` / `Budget too low for any output tokens` warnings. The truncated parseable prefix is a syntactically partial function that scores worse than the implement output, so refine reverts. The two ZEBRA solvers, with $\sim\$13\times10^{-3}$ on refine, log neither marker on seed 1: the revise call completes within budget, the revision is well-formed, and refine selects it.

The mechanism behind the gap is a three-step chain visible seed-by-seed:

1. Implement frequently produces a near-correct but edge-case-buggy solution (the always-add-$+1$ shortcut, the missing `'S' not in trophies` branch, etc.). All five strategies' implement phases share a similar bug rate – mean implement-only score is 0.27–0.31 across LLM, LLM-CoT, ZEBRA-LLM, and `mult_offset`; `additive` is slightly higher at 0.41 thanks to a longer implement window, but still below the pass threshold.
2. Refine *can* fix these bugs – review correctly identifies the edge case on every refine-fire seed we audited – but only if the revise call has enough budget to actually *finish* writing the corrected function. With $\$7$–$9\times10^{-3}$ on refine, gpt-4o exhausts its per-call output-token budget mid-revision and the wrapper logs `[revise] Output missing END_OF_OUTPUT token ...Truncated output to parseable prefix` on the canonical seed-1 logs of LLM-Direct, LLM-CoT, and the ZEBRA-LLM ablation. The truncated prefixes are syntactically incomplete (no

return path, no closing `max` reduction) and score 7–19/182, well below the implement output. With $\$13 \times 10^{-3}$, ZEBRA's revise calls complete cleanly (no truncation marker, no `Budget below preferred length` warning), produce longer revisions ($\sim$35–45 lines vs. $\sim$22–32 lines for the truncated cases), introduce explicit invariants (the `total_g` check on this task), and dominate the implement output on the 182-test grader.

3. ZEBRA's allocators sit in the second regime on every seed (`additive` at $\$13.03 \pm 0.24$, `mult_offset` at $\$13.14 \pm 0.24$, both essentially deterministic across seeds). LLM-based allocators sit in the first regime (LLM-Direct $\$9.08 \pm 0.62$; LLM-CoT $\$6.68 \pm 1.91$; ZEBRA-LLM ablation $\$7.28 \pm 1.89$). The ZEBRA-LLM ablation in particular shows that the win is from the *algorithmic* solver, not the curves – the same curves under an LLM allocator collapse into a refine-starved split that puts the revision in the regression regime.