SQUEEZE EVOLVE
Unified Multi-Model Orchestration for Verifier-Free Evolution

Monishwaran Maheswaran [*1 2] **Leon Lakhani** [*1] **Zhongzhu Zhou** [†2] **Shijia Yang** [†3] **Junxiong Wang** [2] **Coleman Hooper** [1] **Yuezhou Hu** [1] **Rishabh Tiwari** [1] **Jue Wang** [2] **Harman Singh** [1] **Qingyang Wu** [2] **Yuqing Jian** [2] **Ce Zhang** [2] **Kurt Keutzer** [1] **Tri Dao** [4 2] **Xiaoxia Wu** [2] **Ben Athiwaratkun** [2] **James Zou** [‡5 2] **Chenfeng Xu** [*‡3 2]

Abstract

We show that verifier-free evolution is bottlenecked by both diversity and efficiency: without external correction, repeated evolution accelerates collapse toward narrow modes, while the uniform use of a high-cost model wastes compute and quickly becomes economically impractical. We introduce SQUEEZE EVOLVE, a unified multi-model orchestration framework for verifier-free evolutionary inference. Our approach is guided by a simple principle: allocate model capability where it has the highest marginal utility. Stronger models are reserved for high-impact stages, while cheaper models handle the other stages at much lower costs. This principle addresses diversity and cost-efficiency jointly while remaining lightweight. SQUEEZE EVOLVE naturally supports open-source, closed-source, and mixed-model deployments. Across AIME 2025, HMMT 2025, LiveCodeBench V6, GPQA-Diamond, ARC-AGI-V2, and multimodal vision benchmarks, such as MMMU-Pro and BabyVision, SQUEEZE EVOLVE consistently improves the cost–capability frontier over single-model evolution and achieves new state-of-the-art results on several tasks. Empirically, SQUEEZE EVOLVE reduces API cost by up to $\sim$3$\times$ and increases fixed-budget serving throughput by up to $\sim$10$\times$. Moreover, on discovery tasks, SQUEEZE EVOLVE is the first verifier-free evolutionary method to match, and in some cases exceed, the performance of verifier-based evolutionary methods.

[*]Equal contribution. [†]Equal second. [‡]Co-advising. [1]UC Berkeley [2]Together AI [3]UT Austin [4]Princeton University [5]Stanford University. Correspondence to: Monishwaran Maheswaran <monishwaran@berkeley.edu>, Chenfeng Xu <xuchenfeng@utexas.edu>.

The 3rd *AI for Math Workshop* at the 43rd International Conference on Machine Learning (ICML), Seoul, South Korea, 2026.
Copyright 2026 by the author(s).

1. Introduction

Test-time scaling has emerged as a practical way to push language models beyond one-shot inference by spending additional compute at test time to search over or refine candidate solutions (Wang et al., 2023; Madaan et al., 2023; Venkatraman et al., 2026). A particularly promising direction is self-evolution, where models iteratively improve candidates through selection, mutation, and recombination (Novikov et al., 2025; Sharma, 2025; Lange et al., 2025; Liu et al., 2026). When coupled with an external verifier, this paradigm can unlock powerful discovery capabilities. But in many important domains, verification is too expensive and slow, or simply unavailable. For example, in nuclear fusion research, a single tokamak plasma study may require more than 120 million CPU-hours (Howard et al., 2016). This motivates our focus on *verifier-free evolution*.

However, verifier-free evolution is approaching a hard wall: methods such as RSA generate 500–700$\times$ more tokens than single-shot inference, making cost increasingly difficult to sustain. This pressure is compounded by the sharp cost gap between models: proprietary reasoning models remain 4–25$\times$ more expensive per output token than strong hosted open-weight alternatives (OpenAI, 2026; Anthropic, 2026; Google, 2026; Together AI, 2026a;b), and even within the open-weight ecosystem cost differences remain substantial.

As a result, the key question is shifting: rather than only asking *how we can spend more compute and money to unlock new capabilities?*, we must also ask *how we can achieve a given capability target under tight budget constraints?* This is the same principle that has historically driven advances in software and algorithms: progress comes not just from using more resources, but from using them more efficiently and lowering the cost needed to achieve a given capability target.[1] In this work, we advance this principle, as illustrated in Fig. 1.

To answer the above question, we first take a system perspective: many seemingly disparate test-time methods can be expressed as instances of a single evolutionary framework.

[1] https://epochai.substack.com/p/the-least-understood-driver-of-ai

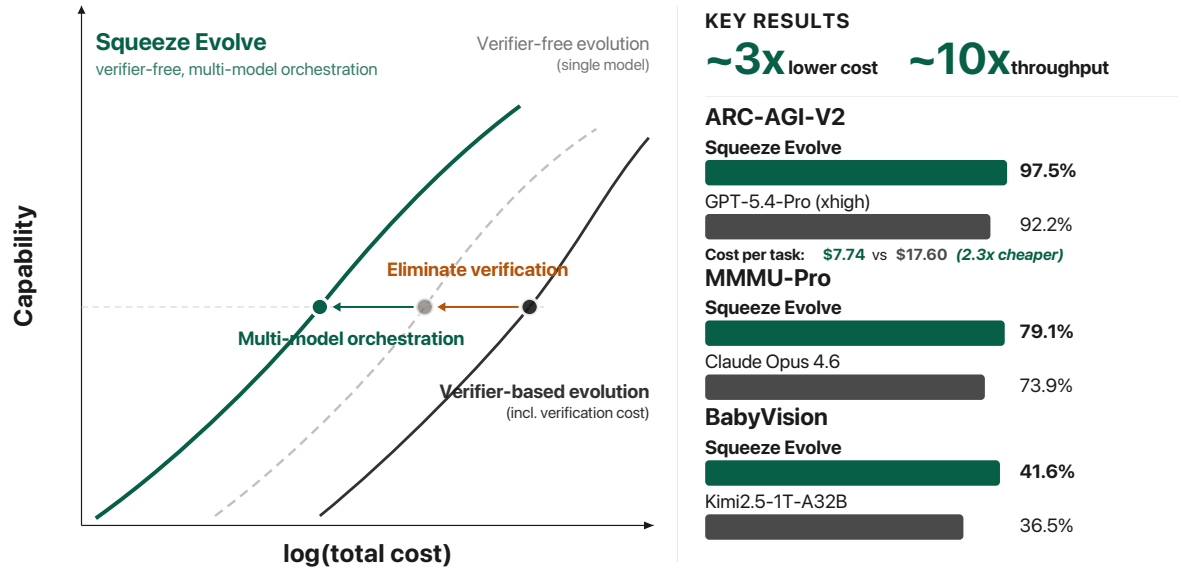


Figure 1. SQUEEZE EVOLVE shifts the cost–capability frontier left by combining verifier-free evolution with multi-model orchestration. **Left:** Conceptual scaling curves. **Right:** Key results across ARC-AGI-V2, MMMU-Pro, and BabyVision.

Once cast in this unified form, they expose a common design space that can be optimized jointly. In Sec. 3, we describe how we unify the current test-time scaling method into a single evolutionary framework, where different operator choices recover a wide spectrum of existing test-time strategies. For example, majority voting (Wang et al., 2023) corresponds to a shallow single-step evolution, recursive self-aggregation (Venkatraman et al., 2026) corresponds to an verifier-free multi-step evolutionary process, and verifier-based self-evolve pipelines such as AlphaEvolve (Novikov et al., 2025) correspond to feedback-driven evolutionary search. Our unified framework naturally highlights the key problems:

1. Given models with different cost–capability trade-offs, which model should be assigned to each operator in the evolutionary pipeline (e.g., initialization, generation, recombination, or fitness estimation)?

2. How should these models be coordinated across the pipeline to maximize capability per unit cost without incurring excessive orchestration overhead?

We answer these two questions through a comprehensive empirical analysis in Section 4. In brief, we find that:

1. **From the verification perspective,** scaling the token budget can partially offset the absence of explicit verification. By spending additional tokens on diverse generation and iterative aggregation, verifier-free evolution can converge reliably toward correct solutions even without external reward signals. This makes verifier-free evolution especially attractive in practice, as it improves capability while avoiding the substantial cost of explicit verification.

2. **From the performance perspective,** unlike verifier-based method, simple verifier-free evolution causes the upper bound (*e.g., pass@N or best continuous score*) to degrade significantly. Such a drop directly limits the achievable performance of the entire pipeline. We further find that this upper bound is highly correlated with generation diversity, highlighting diversity as a central ingredient for effective verifier-free evolution. This further strongly motivates our use of *multi-model orchestration* to preserve diversity and sustain performance.

3. **From the cost perspective,** initialization quality largely determines final accuracy, recombination capability depends on the candidate set being aggregated, and self- and cross-model internal signals serve as reliable fitness proxies. These findings provide a foundation for principled orchestration design.

Motivated by these observations and by the economic mismatch between open and closed model ecosystems, we present SQUEEZE-EVOLVE, a multi-model orchestration framework that routes each evolutionary operation to the most cost-effective model based on confidence-derived fitness signals, reserving expensive models for only the highest-marginal-utility steps.

We evaluate SQUEEZE EVOLVE across AIME 2025, HMMT 2025, GPQA-Diamond, LiveCodeBench V6, MMMU-Pro, BabyVision, ARC-AGI-V2, circle packing, and the agentic Terminal-Bench 2.0, spanning open-source model pairs, mixed open-source and proprietary model pairs, multimodal vision, and agentic settings. In summary,

we make the following contributions:

1. **Unified framework and analysis.** We unify existing test-time scaling methods into a single evolutionary framework, identify the key design axes for multi-model orchestration, and show that diversity collapse is the central bottleneck of verifier-free evolution while model-intrinsic confidence signals serve as effective fitness proxies for routing (Sections 3, 4).

2. **Confidence-based routing.** A lightweight mechanism that assigns each recombination group to the most cost-effective model using only signals already produced during inference (Section 5).

3. **Empirical gains across nine benchmarks.** Across math (AIME 2025, HMMT 2025, GPQA-Diamond), coding (LiveCodeBench V6), vision (MMMU-Pro, BabyVision), visual reasoning (ARC-AGI-V2), and scientific discovery (circle packing), SQUEEZE EVOLVE reduces API cost by 1.3–3.3× while preserving or exceeding single-model accuracy, and in multiple configurations *surpasses* the expensive Model 2 alone (Section 6). On multimodal tasks, a text-only cheap Model 1 that never processes images matches or exceeds the vision-capable Model 2 at 2.3–2.5× savings, showing visual understanding is primarily needed at initialization (Section 6.2). On ARC-AGI-V2, SQUEEZE EVOLVE sets a new state-of-the-art cost–capability frontier of 97.5% at $7.74/task without code execution (Section 6.3); on circle packing, it is the first verifier-free method to match or exceed verifier-based approaches such as AlphaEvolve (Section 6.4). On the agentic Terminal-Bench 2.0 (Team, 2025b), applied as a pure recombination layer on top of an existing GPT-5.4 submission with no re-execution, SQUEEZE EVOLVE lifts pass@1 from 81.80% to 88.76% (+6.96 pp), recovering 99% of the available oracle headroom (Section 6.5).

4. **Co-designed serving.** A custom confidence engine reduces scoring latency by 4–10×, latency-matched GPU pools prevent bottlenecks, end-to-end routing overhead is only 2.4–4.3%, and fixed-budget serving throughput improves by up to ∼10× (Sections 5.2, 7).

2. Related Work

Our work builds on three lines of research (extended discussion in Section 9). Self-aggregation methods (Venkatraman et al., 2026; Wang et al., 2024) use a single model or fixed assignment, leading to diversity collapse (Singh et al., 2026). LLM-driven evolutionary methods (Romera-Paredes et al., 2024; Novikov et al., 2025; Liu et al., 2026) rely on external verifiers and apply one model across all operators. Model routing (Ong et al., 2025; Maheswaran et al., 2025) dispatches queries or steps between models, and token-level confidence has been used to filter traces (Fu et al., 2025). SQUEEZE EVOLVE repurposes confidence for routing within a verifier-free evolutionary pipeline, where savings compound across loops.

3. A Unified Formulation of Evolutionary Framework

We show that many existing methods can be cast within a common evolutionary framework, providing a formal foundation for joint optimization and a simple orchestration rule: invoke the larger model only when the smaller model is likely to exceed its capability limits.

For a query Q, we initialize a population $\mathcal{P}^{(0)}$ using an *ancestor* function p_F, where each candidate $\tau_i^{(0)} \sim p_\theta(\cdot \mid Q)$ is sampled from a generative model θ. Existing methods differ primarily in how they organize, score, and evolve these candidates. We unify these steps into a single *evolutionary operator* Φ_f, which encapsulates selection followed by recombination:

$$\Phi_f(\mathcal{P}) = \mathrm{recomb}_f \circ \mathrm{select}_f(\mathcal{P}), \quad \mathcal{P}^{(t+1)} = \Phi_{f_{t+1}}(\mathcal{P}^{(t)}). \tag{1}$$

This induces an iterated evolutionary process where the final population $\mathcal{P}^{(k)}$ is derived via a sequence of operator compositions:

$$\mathcal{P}^{(k)} = \left(\Phi_{f_k} \circ \Phi_{f_{k-1}} \circ \cdots \circ \Phi_{f_1}\right)(\mathcal{P}^{(0)}), \tag{2}$$

where each operator Φ_{f_i} utilizes the fitness signal f_i to transition between generations. In verifier-free evolution, f is derived entirely from model outputs (e.g., log-probabilities, consensus frequency) without external verification. Table 1 categorizes existing methods by how they instantiate select, recomb, and f.

4. Motivation analysis for verifier-free evolutionary framework

The inherent Pass@K bottleneck of verifier-free evolution. Without an external verifier, evolution can only amplify trajectories the model already recognizes, driving the population to be narrow. As Figure 2 shows across both GPQA-Diamond (Rein et al., 2024) settings, pass@K falls alongside semantic diversity. Multi-model orchestration counteracts this: by combining models with different priors and failure modes, SQUEEZE EVOLVE maintains complementary lineages and stays higher and flatter on both diversity and pass@K.

Ancestor function dominates final accuracy. Using the strong model for initialization and the weak model for recombination consistently outperforms the reverse: GPT-OSS-120B (Agarwal et al., 2025)→20B achieves 0.89 vs. 0.85 on HMMT 2025 (Balunović et al., 2026), and Qwen3-4B-Thinking (Team, 2025a)→Instruct reaches 0.88 vs. 0.65

Table 1. Instantiation of the unified evolutionary framework for test-time scaling methods.

Method	k	$\lvert\mathcal{P}\rvert$	select	recombination	Fitness f	Model
Majority Voting	1	N	Answer clustering	Identity	Consensus frequency	Single
Self-Refinement	T	1	Self-critique	Conditioned rewrite	Natural language critique	Single
RSA	T	N	K-subset	LLM aggregation	Implicit	Single
AlphaEvolve	T	Variable	Fitness-guided	LLM aggregation	External h	Multi-model
SQUEEZE EVOLVE	T	Variable	Fitness-guided	Mix of recombination	Probabilistic fitness	Multi-model

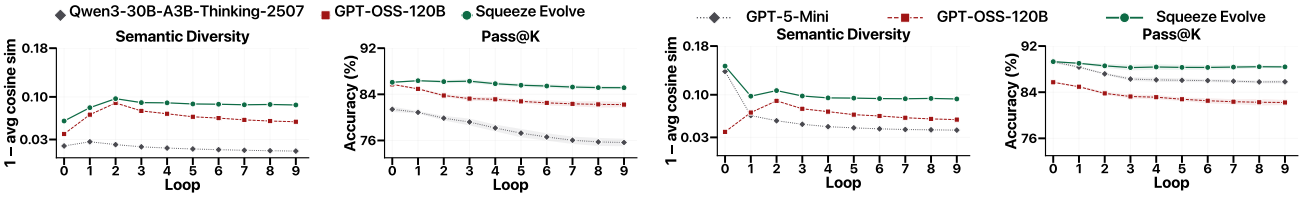


Figure 2. **Single-model evolution collapses diversity and shrinks pass@K; multi-model routing preserves both.** Single-model baselines lose diversity after the early loops with a corresponding decline in pass@K, whereas SQUEEZE EVOLVE remains higher and flatter on both metrics.

Table 2. **Ancestor function dominates final accuracy.** Mean final loop (9) accuracy across 4 seeds. Strong-init → weak-agg outperforms weak-init → strong-agg, indicating initialization quality dominates aggregation quality.

Model pair	Data	S→W	W→S	Δ
GPT-OSS-120B / GPT-OSS-20B	HMMT'25	0.89	0.85	+4
Qwen3-4B-Thinking / Qwen3-4B-Instruct	AIME'25	0.88	0.65	+23

on AIME 2025 (Table 2). This asymmetry motivates reserving the strong model for initialization.

Weak models suffice when candidates are strong. Recombination accuracy rises from 0% with no correct candidate to 100% when all four are correct (Fig. 3a), consistently across benchmarks and model families. This motivates routing high-quality subsets to a cheaper model.

Confidence as a fitness proxy. Both self- and cross-model confidence reliably separate high-quality from low-quality subsets (Figure 3b), motivating their use as the fitness signal for routing.

5. Squeeze Evolve

We instantiate Φ_f from Section 3 as a single algorithm (Figure 4; Algorithm 1, Section 13). The key extension is fitness-based routing: each candidate group is assigned to one of N tiers, $\text{Model}_{1..N-1}$ (ordered by increasing cost) or lightweight aggregation, based on the fitness signal (Table 3).

5.1. Algorithm

Each loop scores candidates via the fitness signal f, applies select_f to form groups, routes each group to one of three recombination tiers within recomb_f, and updates the population. We define each component below.

Initialization. The initial population is generated entirely by the target model: $\mathcal{P}_q^{(0)} = \{\tau_i \sim p_{M_2}(\cdot \mid Q_q)\}_{i=1}^{N}$. This is motivated by the finding that initialization quality is the single strongest predictor of final accuracy (Table 2).

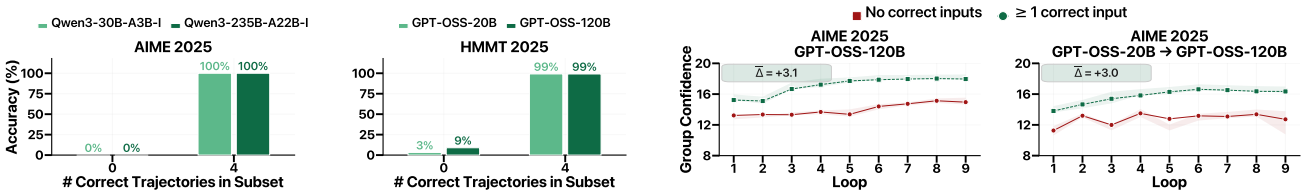
Fitness signal. SQUEEZE EVOLVE uses two model-intrinsic fitness signals, both proxies for group difficulty. *Group confidence* (GC) averages per-token confidence over a trajectory and then over the group:

$$\begin{aligned} c(i) &= -\tfrac{1}{K_\ell}\textstyle\sum_{j=1}^{K_\ell} \log p_\theta\big(v_j^{(i)} \mid t_{<i}, Q\big), \\ C(\tau) &= \tfrac{1}{|\tau|}\textstyle\sum_i c(i), \quad \mathrm{GC}(g) = \tfrac{1}{K}\textstyle\sum_{\tau\in g} C(\tau). \end{aligned} \tag{3}$$

where $\{v_1^{(i)}, \ldots, v_{K_\ell}^{(i)}\}$ are the K_ℓ most likely tokens under scoring model θ. The per-token confidence $c(i)$ follows the same formulation used by (Fu et al., 2025) to filter reasoning traces; here we aggregate it to the group level for routing. When θ is the generator itself this is *self-confidence* (zero cost); otherwise it is *cross-model confidence* (one prefill pass per candidate; Section 5.2). When log-probabilities are unavailable, we instead use *group diversity* $D(g) = \big|\{\text{answer}(\tau) : \tau \in g\}\big|$, the number of unique final answers. The choice between the two signals is determined by API access.

Selection. At each loop $t \geq 1$, we form M groups of size K from the current population. Groups can be formed by *uniform sampling* (random K-subsets or by *fitness-weighted sampling*, where candidates are drawn with probability $\exp\big(f(\tau_i)/\zeta\big) / \sum_j \exp\big(f(\tau_j)/\zeta\big)$ and a temperature ζ controls the exploitation–exploration balance.)

Recombination. Groups with sufficient consensus are aggregated via lightweight methods ($\mathcal{B}_{\text{lite}}$). The remain-



(a) Aggregation accuracy at the extremes: 0 vs. 4 correct trajectories.

(b) Group confidence separates correct from incorrect trajectories.

Figure 3. **Aggregation success is seed-dependent, and group confidence predicts it. (a)** With zero correct seeds neither model recovers a correct answer; with all four correct both reach near-perfect accuracy (full 0–4 breakdown in Fig. 10). **(b)** Mean group confidence across loops 1–9 on AIME 2025: correct-containing subsets maintain consistently higher GC ($\overline{\Delta} \geq +3.0$) in both self- and cross-model settings (Figs. 11, 12).

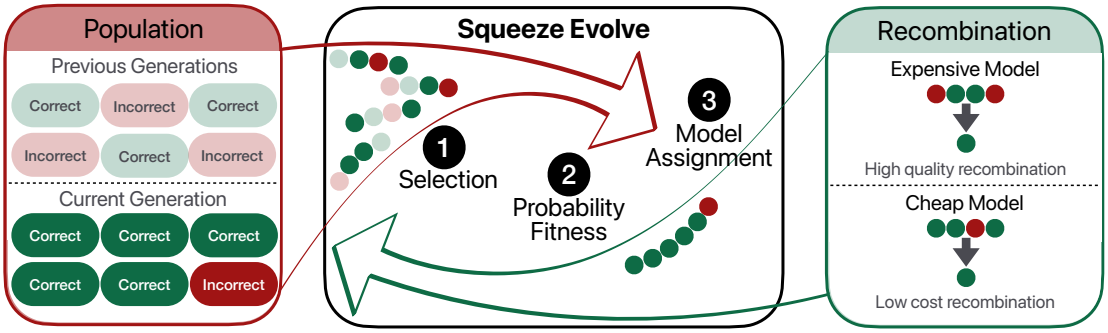


Figure 4. SQUEEZE EVOLVE overview. The expensive Model 2 generates the initial population; subsequent loops recombine groups using Model 1 and 2 based on group confidence

ing groups are split by a per-problem adaptive threshold $\theta_q = \mathrm{Percentile}_p\big(\{F(g) : g \in \mathcal{G}_q \setminus \mathcal{B}_{\mathrm{lite}}\}\big)$: groups that are "easy" under the fitness signal (high GC or low D) are routed to Model 1, and the rest to Model 2. The routing percentile p is the single deployment-time hyperparameter. The two model batches execute in parallel, and trajectories from all three tiers are merged via either *replace* or *accumulate* ($\mathcal{P}_q^{(t)} = \mathcal{P}_q^{(t-1)} \cup \mathcal{R}_{\mathrm{new}}$).

5.2. System Implementation

Self-confidence is free: the generating model already produces the token log-probabilities needed to score its own trajectory. *Cross-model confidence* requires a single prefill-only forward pass per trajectory under a different model; because that model is already resident for its aggregation branch, this adds no memory overhead, and the N scoring calls per loop are batched into a single request.

Standard serving systems (Kwon et al., 2023; Zheng et al., 2024) are optimized for decode-heavy generation, but confidence scoring needs just one scalar per trajectory. To avoid materializing full token-level logprob tensors, we implement a custom prefill path in vLLM that accumulates the confidence statistic directly on GPU and returns only the final scalar, reducing per-request transfer from ∼13 MB to ∼100 bytes. This achieves 4–10× lower scoring latency and enables confidence scoring on Qwen3-235B-A22B where the native path runs out of memory (Section 20). We quantify end-to-end routing overhead and system throughput in Section 7.

6. Evaluation

All runs use population N=16, group size K=4, and T=10 evolutionary loops, averaged over four seeds, unless stated otherwise. Costs are measured in actual API dollars per problem using model provider pricing (Table 8; generation hyperparameters in Table 7). The baseline is standard RSA with Model 2 only, which serves as the cost upper bound.

6.1. Math and Coding

We evaluate SQUEEZE EVOLVE on AIME 2025, HMMT 2025 (Balunović et al., 2026), GPQA-Diamond (Rein et al., 2024), and LiveCodeBench V6 (Jain et al., 2024). Accuracy-vs-cost curves appear in Figures 5 and 6; full per-percentile breakdowns in Tables 9 and 10.

Homogeneous pairs (open-source). We test three pairs spanning different design axes: Qwen3-30B Instruct / Thinking (same scale, different mode), Qwen3-30B / 235B Instruct (different scale, same mode), and GPT-OSS-20B / 120B (different scale, both thinking). SQUEEZE EVOLVE matches or exceeds Model 2 alone at **1.4–2.1× less** cost, surpassing Model 2 on two of three pairs (+1.5 pp on AIME25 for Instruct→Thinking; +2.3 pp

Table 3. Operator instantiations of SQUEEZE EVOLVE across evaluation settings.

Setting	Fitness f	Select_f	Recomb_f (routing rule)	Update
Math / Coding / Vision	GC (Eq. 3)	Uniform	Percentile on GC	Replace
ARC-AGI-V2 (§6.3)	Diversity D (Eq. 5.1)	Uniform	Threshold on D + lite agg	Replace
Circle Packing (§6.4)	GC (Eq. 3)	Fitness-weighted (ζ=0.5)	Percentile on GC	Accumulate

on HMMT25 for GPT-OSS). Even with Model 1 much smaller (30B vs. 235B), accuracy stays within 1 point at nearly half the cost; the GPT-OSS pair matches Model 2 on LiveCodeBench V6 at 2.0× savings.

Heterogeneous pairs (open-source + closed-source). We pair Qwen3-30B Instruct and GPT-OSS-20B with GPT-5 mini (OpenAI, 2025) as Model 2, sweeping $p \in \{0, 10, 20, 30\}$ (Model 1 scores via prefill since GPT-5 mini does not expose output logprobs; this cost is included). SQUEEZE EVOLVE achieves **1.4–3.3× savings**: at p=30, GPT-OSS-20B + GPT-5 mini exceeds Model 2 on AIME25 (95.4% vs. 94.2%) at 1.8× savings; at p=0, savings reach 3× with accuracy drops of only 1.5–6 pp (Table 10).

6.2. Multimodal Vision Task

We evaluate SQUEEZE EVOLVE on two multimodal benchmarks: MMMU-Pro (Yue et al., 2025) and BabyVision (Chen et al., 2026), using T=5 loops (other settings match Section 6.1). We test a homogeneous pair (Kimi-2.5 Instant / Thinking (Team et al., 2026), both vision-capable) and a heterogeneous pair (Qwen3.5-35B (Qwen Team, 2026) → Kimi-2.5 Thinking, where Model 1 operates in *text-only mode* after loop 0).

Accuracy-vs-cost curves for MMMU-Pro appear in Figure 8. On MMMU-Pro, the homogeneous pair matches Model 2 at **1.9**× savings, while the heterogeneous pair *surpasses* Model 2 (79.1% vs. 78.6%) at **2.7**× savings, even though Model 1 never sees any images. On BabyVision, the homogeneous pair preserves accuracy at **2.5**× savings. The heterogeneous result further reinforces the finding from Section 4 that initialization quality is the dominant factor: once loop 0 grounds the population in image content, subsequent aggregation can be delegated to a cheaper text-only model. Full breakdowns appear in Tables 11 and 12.

6.3. ARC-AGI-V2

We evaluate SQUEEZE EVOLVE on ARC-AGI-V2 (Chollet et al., 2025) public evaluation set. Since the Gemini API does not expose logprobs, we use answer diversity (Eq. 5.1) as the fitness signal (Table 3). Groups with non-zero diversity are recombined by Gemini 3.1 Pro (Google DeepMind, 2026); consensus groups fall back to majority vote. With this routing, SQUEEZE EVOLVE achieves 97.5% at $7.74/task.

Using this result as a baseline, we further reduce cost by adding Gemini 3.0 Flash (Google DeepMind, 2025) as Model 1 to the recombination function, yielding a three-way routing rule: high-diversity groups with 3 or more unique answers invoke the expensive Gemini 3.1 Pro Model 2, lower diversity groups are handled by Gemini 3.0 Flash, and groups that have already reached consensus are aggregated via lightweight non-LLM methods (e.g., majority vote). With this recombination function, we observe immediate convergence to the pass@k score after one aggregation step, achieving the same 97.5% accuracy for only $5.93/task, a 1.2× savings. This is a new SoTA cost-capability frontier result on ARC-AGI-V2 public evaluation set to date. Even compared to code-execution-based approaches, SQUEEZE EVOLVE reaches comparable accuracy at a lower cost to Confluence Lab (Confluence Labs, 2026) (97.9%, $11.77/task) and Imbue (Imbue, 2026) (95.1%, $8.71/task).

Table 4. ARC-AGI-V2 public evaluation results. N=4, K=2. †Uses code execution and program synthesis. Extended results in Table 16.

Strategy	Model 1	Model 2	Acc.	$/Task	Savings
Code-execution methods					
Imbue†	—	Gemini 3.1 Pro	95.1	$8.71	—
Confluence Lab†	—	—	97.9	$11.77	—
Full pipeline (T=10)					
RSA	—	Gemini 3.1 Pro	93.3	$28.85	1.0×
SQUEEZE EVOLVE	—	Gemini 3.1 Pro	**97.5**	$7.74	3.7×
Single recombination (T=2)					
SQUEEZE EVOLVE	—	Gemini 3.1 Pro	94.2	$5.62	5.1×
SQUEEZE EVOLVE	Gemini 3.0 Flash	Gemini 3.1 Pro	**97.5**	**$5.93**	**4.9**×

Table 5. Comparison of methods on Circle Packing (n=26, ↑). ShinkaEvolve uses an ensemble of Claude Sonnet-4, GPT-4.1, GPT-4.1-mini, GPT-4.1-nano, and o4-mini.

Method	Model	Score (↑)
ShinkaEvolve (Lange et al., 2025)	Ensemble (see caption)	2.635982
SQUEEZE EVOLVE	GPT-OSS-120B + 20B	2.635896
AlphaEvolve (Novikov et al., 2025)	Gemini-2.0 Pro + Flash	2.635862
OpenEvolve (Sharma, 2025)	Gemini-2.0 Flash + Claude 3.7 Sonnet	2.634292

6.4. Circle Packing: Scientific Discovery

We apply SQUEEZE EVOLVE to the circle packing problem from AlphaEvolve (Novikov et al., 2025): pack $n = 26$ non-overlapping circles in a unit square to maximize the sum of their radii. This is an open-ended optimization with a continuous objective. We use GPT-OSS-20B as Model 1

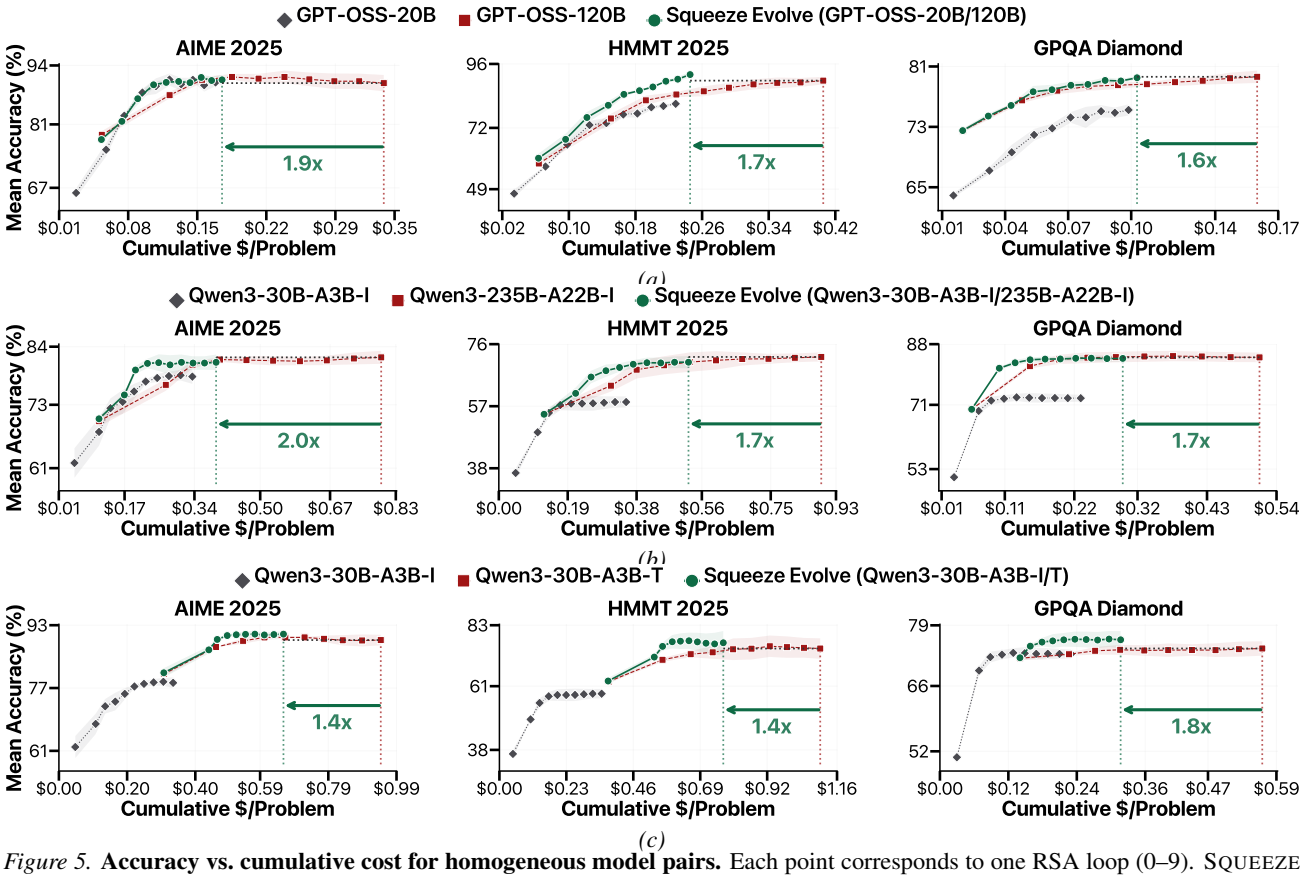


Figure 5. **Accuracy vs. cumulative cost for homogeneous model pairs.** Each point corresponds to one RSA loop (0–9). SQUEEZE EVOLVE (green) tracks the RSA accuracy curve while staying significantly further left, achieving comparable quality at 1.4–2.0× lower cost.

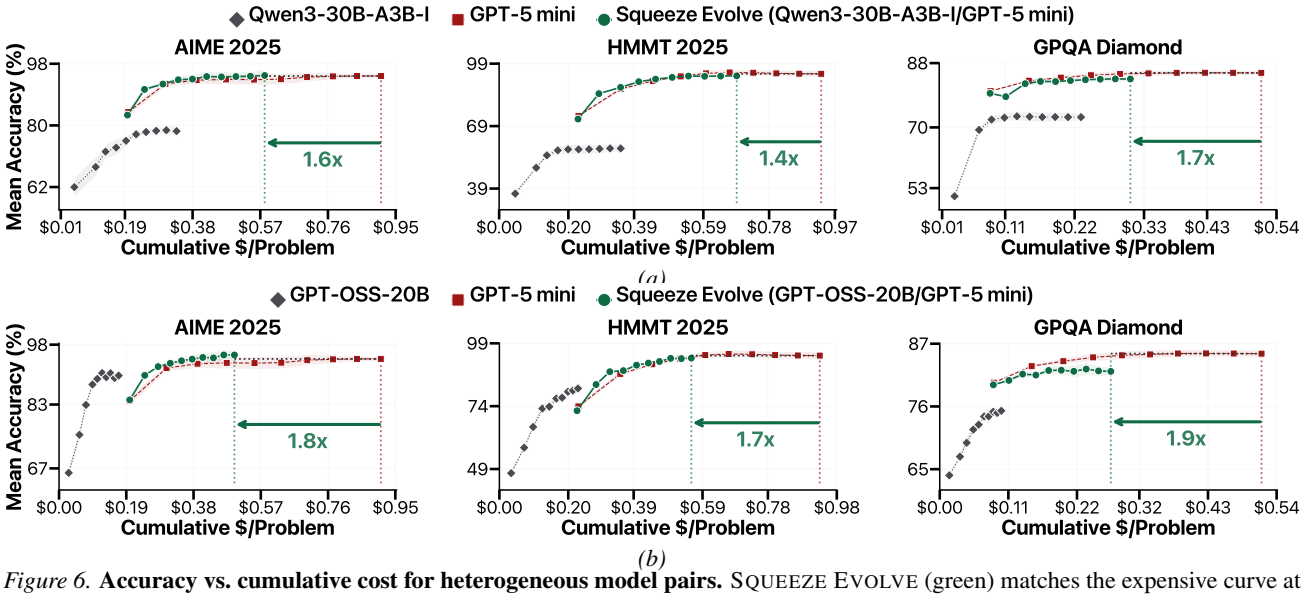


Figure 6. **Accuracy vs. cumulative cost for heterogeneous model pairs.** SQUEEZE EVOLVE (green) matches the expensive curve at 1.4–1.9× lower cost, demonstrating that confidence-based routing generalizes across model families and access types.

and GPT-OSS-120B as Model 2; operator choices appear in Section 24.2. SQUEEZE EVOLVE is competitive with leading evolutionary frameworks (Table 5) without executing generated programs or using closed-source models. Unlike other frameworks that feed execution results back into generation, SQUEEZE EVOLVE relies solely on model-

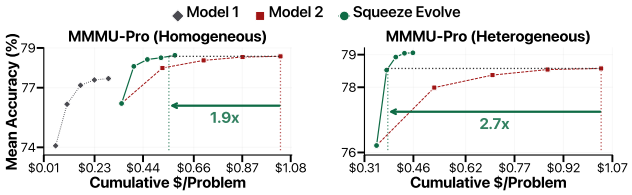


Figure 8. **Accuracy vs. cumulative cost on MMMU-Pro for homogeneous and heterogeneous vision pairs.** Left: Kimi-2.5 Instant (Model 1) / Thinking (Model 2). Right: Qwen3.5-35B (Model 1, text-only) → Kimi-2.5 Thinking (Model 2). Savings are measured at matched accuracy. The heterogeneous pair achieves 2.7× savings despite Model 1 never seeing any images.

intrinsic confidence, which correlates weakly but sufficiently with the objective to improve both average and best scores over iterations. Details and discovered program appear in Section 24.

6.5. Terminal-Bench: Agentic Setting

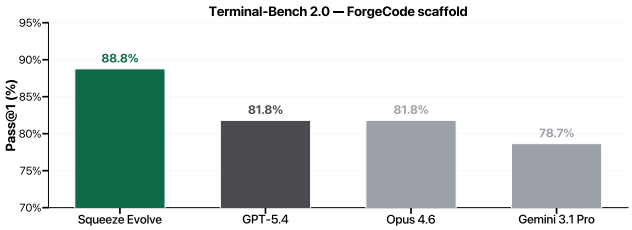


Figure 9. Terminal-Bench 2.0 pass@1 on the ForgeCode scaffold. SQUEEZE EVOLVE lifts pass@1 from 81.80% to 88.76% (+6.96 pp) by recombining existing GPT-5.4 trials, without re-running the agent.

We apply SQUEEZE EVOLVE to Terminal-Bench 2.0 (Team, 2025b): 89 software-engineering tasks in sandboxed terminals with deterministic pass/fail checks. We treat the five public ForgeCode / GPT-5.4 trials per task as $\mathcal{P}_q^{(0)}$ and apply a single recombination step with Gemini 3 Flash as Model 2; the operator reads each trajectory's exit flags and stdout/stderr tail, and never re-executes in the sandbox. SQUEEZE EVOLVE lifts pass@1 from 81.80% to 88.76% (**+6.96 pp**) over the public leaderboard (Figure 9), without re-running the agent. Unlike approaches that scale agentic settings by spawning more rollouts, SQUEEZE EVOLVE reuses existing trajectories: of the 80 tasks with at least one passing trial (oracle pass@5 = 89.89%), recombination emits a passer in 79, recovering 99% of the available headroom.

7. System Results

Routing overhead. A natural systems question is whether confidence scoring and model dispatch introduce enough additional latency to undermine multi-model routing. To isolate this cost, we compare RSA with Model 2 (no routing) against SQUEEZE EVOLVE with Model 2 (routing enabled but all aggregation forced to Model 2), so the only difference is the routing logic itself. Figure 16 shows that routing adds only 2.4–4.3% to total latency across all three target models, confirming that confidence scoring is a batched prefill-only operation whose cost is negligible relative to generation (full details in Section 21).

System throughput. We next ask whether routing improves steady-state throughput under a fixed GPU budget G. RSA assigns all G to Model 2, while SQUEEZE EVOLVE splits them into Model 2 and Model 1 pools ($G_2{+}G_1{=}G$) with matched loop service times: under-provisioning either side bottlenecks the system and erases the gains from cheaper aggregation. We report requests per second because the two models emit different output lengths. Figure 7 shows Qwen3-30B/235B reaches **4–10**× speedup and GPT-OSS-20B/120B **1.4–3.4**×; the gap tracks Qwen's larger asymmetry on both total and active parameters (full breakdowns in Section 22).

8. Conclusion

SQUEEZE EVOLVE uses confidence-based routing to make verifier-free evolutionary inference cheaper without sacrificing quality: across math, coding, vision, agentic, and discovery tasks, it cuts cost 1.4–3.3×, sets a new SoTA on ARC-AGI-V2, matches verifier-based methods on circle packing, and yields up to 10× throughput gains.

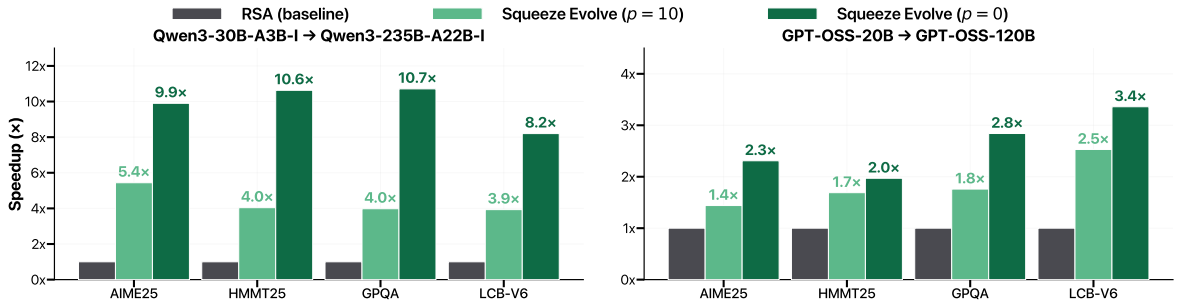


Figure 7. **Fixed-budget throughput speedup over RSA.** Under the same total GPU budget, the Qwen pair achieves 4–10× speedup and the GPT-OSS pair 1.4–3.4×. Full results in Table 15.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Agarwal, S., Ahmad, L., Ai, J., Altman, S., Applebaum, A., Arbus, E., Arora, R. K., Bai, Y., Baker, B., and et al., H. B. gpt-oss-120b & gpt-oss-20b model card, 2025. URL https://arxiv.org/abs/2508.10925.

Agrawal, L. A., Tan, S., Soylu, D., Ziems, N., Khare, R., Opsahl-Ong, K., Singhvi, A., Shandilya, H., Ryan, M. J., Jiang, M., Potts, C., Sen, K., Dimakis, A. G., Stoica, I., Klein, D., Zaharia, M., and Khattab, O. Gepa: Reflective prompt evolution can outperform reinforcement learning, 2026. URL https://arxiv.org/abs/2507.19457.

Anthropic. Pricing. https://platform.claude.com/docs/en/about-claude/pricing, 2026. Claude API pricing page, accessed March 16, 2026.

Assumpção, H., Ferreira, D., Campos, L., and Murai, F. Codeevolve: an open source evolutionary coding agent for algorithmic discovery and optimization, 2026. URL https://arxiv.org/abs/2510.14150.

Balunović, M., Dekoninck, J., Petrov, I., Jovanović, N., and Vechev, M. Matharena: Evaluating llms on uncontaminated math competitions, 2026. URL https://arxiv.org/abs/2505.23281.

Bansal, H., Hosseini, A., Agarwal, R., Tran, V. Q., and Kazemi, M. Smaller, weaker, yet better: Training llm reasoners via compute-optimal sampling, 2024. URL https://arxiv.org/abs/2408.16737.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Cemri, M., Agrawal, S., Gupta, A., Liu, S., Cheng, A., Mang, Q., Naren, A., Erdogan, L. E., Sen, K., Zaharia, M., Dimakis, A., and Stoica, I. Adaevolve: Adaptive llm driven zeroth-order optimization, 2026. URL https://arxiv.org/abs/2602.20133.

Chen, L., Xie, W., Liang, Y., He, H., Zhao, H., Yang, Z., Huang, Z., Wu, H., Lu, H., charles, Y., Bao, Y., Fan, Y., Li, G., Shen, H., Chen, X., Xu, W., Si, S., Cai, Z., Chai, W., Huang, Z., Liu, F., Liu, T., Chang, B., Hu, X., Chen, K., Ren, Y., Liu, Y., Gong, Y., and Li, K. Babyvision: Visual reasoning beyond language, 2026. URL https://arxiv.org/abs/2601.06521.

Chollet, F., Knoop, M., Kamradt, G., and Landers, C. ARC-AGI-2: A new challenge for frontier AI reasoning systems. *arXiv preprint arXiv:2505.11831*, 2025.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Confluence Labs. State-of-the-art ARC-AGI-2 solver, 2026. URL https://github.com/confluence-labs/arc-agi-2. GitHub repository, accessed March 2026.

Fu, Y., Wang, X., Tian, Y., and Zhao, J. Deep think with confidence, 2025. URL https://arxiv.org/abs/2508.15260.

Google. Gemini developer api pricing. https://ai.google.dev/gemini-api/docs/pricing, 2026. Google AI for Developers pricing page, accessed March 16, 2026.

Google DeepMind. Gemini 3 Flash model card. Technical report, Google DeepMind, December 2025. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-Flash-Model-Card.pdf.

Google DeepMind. Gemini 3.1 Pro model card. Technical report, Google DeepMind, February 2026. URL https://storage.googleapis.com/deepmind-media/Model-Cards/Gemini-3-1-Pro-Model-Card.pdf.

Guo, D., Yang, D., Zhang, H., Song, J., Wang, P., Zhu, Q., Xu, R., Zhang, R., Ma, S., and Bi, X. e. a. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645 (8081):633–638, September 2025. ISSN 1476-4687. doi: 10.1038/s41586-025-09422-z. URL http://dx.doi.org/10.1038/s41586-025-09422-z.

Howard, N. T., Holland, C., White, A. E., Greenwald, M., and Candy, J. Multi-scale gyrokinetic simulation of tokamak plasmas: enhanced heat loss due to cross-scale coupling of plasma turbulence. *Nuclear Fusion*, 56, 2016.

Imbue. Beating ARC-AGI-2 with code evolution, 2026. URL https://imbue.com/research/2026-02-27-arc-agi-2-evolution/. Blog post, accessed March 2026.

Jaech, A., Kalai, A., Lerer, A., Richardson, A., El-Kishky, A., Low, A., Helyar, A., Madry, A., Beutel, A., and et al., A. C. Openai o1 system card, 2024. URL https://arxiv.org/abs/2412.16720.

Jain, N., Han, K., Gu, A., Li, W.-D., Yan, F., Zhang, T., Wang, S., Solar-Lezama, A., Sen, K., and Stoica, I. Livecodebench: Holistic and contamination free evaluation of large language models for code, 2024. URL https://arxiv.org/abs/2403.07974.

Khairi, A., D'souza, D., Fadaee, M., and Kreutzer, J. Making, not taking, the best of n, 2025. URL https://arxiv.org/abs/2510.00931.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Lange, R. T., Imajuku, Y., and Cetin, E. Shinkaevolve: Towards open-ended and sample-efficient program evolution, 2025. URL https://arxiv.org/abs/2509.19349.

Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C., and Stanley, K. O. Evolution through large models, 2022. URL https://arxiv.org/abs/2206.08896.

Li, Z., Feng, X., Cai, Y., Zhang, Z., Liu, T., Liang, C., Chen, W., Wang, H., and Zhao, T. Llms can generate a better answer by aggregating their own responses, 2025. URL https://arxiv.org/abs/2503.04104.

Lightman, H., Kosaraju, V., Burda, Y., Edwards, H., Baker, B., Lee, T., Leike, J., Schulman, J., Sutskever, I., and Cobbe, K. Let's verify step by step, 2023. URL https://arxiv.org/abs/2305.20050.

Liu, S., Agarwal, S., Maheswaran, M., Cemri, M., Li, Z., Mang, Q., Naren, A., Boneh, E., Cheng, A., Pan, M. Z., Du, A., Keutzer, K., Cheung, A., Dimakis, A. G., Sen, K., Zaharia, M., and Stoica, I. Evox: Meta-evolution for automated discovery, 2026. URL https://arxiv.org/abs/2602.23413.

Lu, J., Teehan, R., Jin, J., and Ren, M. When does verification pay off? a closer look at llms as solution verifiers, 2025. URL https://arxiv.org/abs/2512.02304.

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Madaan, L., Didolkar, A., Gururangan, S., Quan, J., Silva, R., Salakhutdinov, R., Zaheer, M., Arora, S., and Goyal, A. Rethinking thinking tokens: Llms as improvement operators, 2025. URL https://arxiv.org/abs/2510.01123.

Maheswaran, M., Tiwari, R., Hu, Y., Dilmen, K., Hooper, C., Xi, H., Lee, N., Farajtabar, M., Mahoney, M. W., Keutzer, K., and Gholami, A. Arbitrage: Efficient reasoning via advantage-aware speculation, 2025. URL https://arxiv.org/abs/2512.05033.

Muennighoff, N., Yang, Z., Shi, W., Li, X. L., Fei-Fei, L., Hajishirzi, H., Zettlemoyer, L., Liang, P., Candès, E., and Hashimoto, T. s1: Simple test-time scaling, 2025. URL https://arxiv.org/abs/2501.19393.

Novikov, A., Vũ, N., Eisenberger, M., Dupont, E., Huang, P.-S., Wagner, A. Z., Shirobokov, S., Kozlovskii, B., Ruiz, F. J. R., Mehrabian, A., Kumar, M. P., See, A., Chaudhuri, S., Holland, G., Davies, A., Nowozin, S., Kohli, P., and Balog, M. Alphaevolve: A coding agent for scientific and algorithmic discovery, 2025. URL https://arxiv.org/abs/2506.13131.

Ong, I., Almahairi, A., Wu, V., Chiang, W.-L., Wu, T., Gonzalez, J. E., Kadous, M. W., and Stoica, I. Routellm: Learning to route llms with preference data, 2025. URL https://arxiv.org/abs/2406.18665.

OpenAI. GPT-5 system card. Technical report, OpenAI, August 2025. URL https://cdn.openai.com/gpt-5-system-card.pdf.

OpenAI. o4-mini model. https://developers.openai.com/api/docs/models/o4-mini, 2026. OpenAI API model page, accessed March 16, 2026.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026. URL https://qwen.ai/blog?id=qwen3.5.

Rein, D., Hou, B. L., Stickland, A. C., Petty, J., Pang, R. Y., Dirani, J., Michael, J., and Bowman, S. R. Gpqa: A graduate-level google-proof q&a benchmark. In *First Conference on Language Modeling*, 2024.

Romera-Paredes, B., Barekatain, M., Novikov, A., Balog, M., Kumar, M. P., Dupont, E., Ruiz, F. J. R., Ellenberg, J. S., Wang, P., Fawzi, O., Kohli, P., and Fawzi, A. Mathematical discoveries from program search with large language models. *Nature*, 625 (7995):468–475, 2024. doi: 10.1038/s41586-023-06924-6. URL https://doi.org/10.1038/s41586-023-06924-6.

Setlur, A., Rajaraman, N., Levine, S., and Kumar, A. Scaling test-time compute without verification or rl is suboptimal, 2025. URL https://arxiv.org/abs/2502.12118.

Sharma, A. Openevolve: an open-source evolutionary coding agent, 2025. URL https://github.com/algorithmicsuperintelligence/openevolve.

Singh, H., Li, X., Sareen, K., Maheswaran, M., Tan, S., Wu, X., Wang, J., Ariyak, A., Wu, Q., Khaki, S., Tiwari, R., Lian, L., Lu, Y., Li, B., Suhr, A., Athiwaratkun, B., and Keutzer, K. v_1: Unifying generation and self-verification for parallel reasoners, 2026. URL https://arxiv.org/abs/2603.04304.

Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Team, K., Bai, T., Bai, Y., Bao, Y., Cai, S. H., Cao, Y., Charles, Y., Che, H. S., Chen, C., Chen, G., and et al., H. C. Kimi k2.5: Visual agentic intelligence, 2026. URL https://arxiv.org/abs/2602.02276.

Team, Q. Qwen3 technical report, 2025a. URL https://arxiv.org/abs/2505.09388.

Team, T. T.-B. Terminal-bench: A benchmark for ai agents in terminal environments, Apr 2025b. URL https://github.com/laude-institute/terminal-bench.

Together AI. gpt-oss-120b api. https://www.together.ai/models/gpt-oss-120b, 2026a. Together AI model page, accessed March 16, 2026.

Together AI. Qwen3 235b a22b instruct 2507 fp8 api. https://www.together.ai/models/qwen3-235b-a22b-instruct-2507-fp8, 2026b. Together AI model page, accessed March 16, 2026.

Valkanas, A., Pal, S., Rumiantsev, P., Zhang, Y., and Coates, M. C3po: Optimized large language model cascades with probabilistic cost constraints for reasoning, 2025. URL https://arxiv.org/abs/2511.07396.

Venkatraman, S., Jain, V., Mittal, S., Shah, V., Obando-Ceron, J., Bengio, Y., Bartoldson, B. R., Kailkhura, B., Lajoie, G., Berseth, G., Malkin, N., and Jain, M. Recursive self-aggregation unlocks deep thinking in large language models, 2026. URL https://arxiv.org/abs/2509.26626.

Wang, J., Wang, J., Athiwaratkun, B., Zhang, C., and Zou, J. Mixture-of-agents enhances large language model capabilities, 2024. URL https://arxiv.org/abs/2406.04692.

Wang, X., Wei, J., Schuurmans, D., Le, Q., Chi, E., Narang, S., Chowdhery, A., and Zhou, D. Self-consistency improves chain of thought reasoning in language models, 2023. URL https://arxiv.org/abs/2203.11171.

Weng, Y., Zhu, M., Xia, F., Li, B., He, S., Liu, S., Sun, B., Liu, K., and Zhao, J. Large language models are better reasoners with self-verification, 2023. URL https://arxiv.org/abs/2212.09561.

Wu, Y., Sun, Z., Li, S., Welleck, S., and Yang, Y. Inference scaling laws: An empirical analysis of compute-optimal inference for problem-solving with language models, 2025. URL https://arxiv.org/abs/2408.00724.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL https://arxiv.org/abs/2305.10601.

Yue, X., Zheng, T., Ni, Y., Wang, Y., Zhang, K., Tong, S., Sun, Y., Yu, B., Zhang, G., Sun, H., Su, Y., Chen, W., and Neubig, G. Mmmu-pro: A more robust multi-discipline multimodal understanding benchmark, 2025. URL https://arxiv.org/abs/2409.02813.

Yuksekgonul, M., Koceja, D., Li, X., Bianchi, F., McCaleb, J., Wang, X., Kautz, J., Choi, Y., Zou, J., Guestrin, C., and Sun, Y. Learning to discover at test time, 2026. URL https://arxiv.org/abs/2601.16175.

Zhang, D., Huang, X., Zhou, D., Li, Y., and Ouyang, W. Accessing gpt-4 level mathematical olympiad solutions via monte carlo tree self-refine with llama-3 8b, 2024. URL https://arxiv.org/abs/2406.07394.

Zhang, L., Hosseini, A., Bansal, H., Kazemi, M., Kumar, A., and Agarwal, R. Generative verifiers: Reward modeling as next-token prediction, 2025. URL https://arxiv.org/abs/2408.15240.

Zheng, L., Yin, L., Xie, Z., Sun, C., Huang, J., Yu, C. H., Cao, S., Kozyrakis, C., Stoica, I., Gonzalez, J. E., Barrett, C., and Sheng, Y. Sglang: Efficient execution of structured language model programs, 2024. URL https://arxiv.org/abs/2312.07104.

Zhou, A., Yan, K., Shlapentokh-Rothman, M., Wang, H., and Wang, Y.-X. Language agent tree search unifies reasoning acting and planning in language models, 2024. URL https://arxiv.org/abs/2310.04406.

9. Related Work

Test-time scaling. Test-time scaling invests additional inference compute to improve output quality (Snell et al., 2024; Wu et al., 2025), through parallel sampling (Wang et al., 2023; Brown et al., 2024), sequential refinement (Madaan et al., 2023; Muennighoff et al., 2025), search (Yao et al., 2023; Zhang et al., 2024; Zhou et al., 2024), or extended reasoning chains (Jaech et al., 2024; Guo et al., 2025). Compute-optimal sampling with weaker models can outperform a single strong model (Bansal et al., 2024), though scaling without verification remains suboptimal (Setlur et al., 2025). All of these operate within a single-model regime; SQUEEZE EVOLVE extends test-time scaling to multi-model orchestration by routing evolutionary operations across models of different cost.

Self-aggregation and recursive refinement. Several methods combine multiple LLM outputs into a refined answer, including RSA (Venkatraman et al., 2026), generative self-aggregation (Li et al., 2025), Parallel-Distill-Refine (Madaan et al., 2025), and Best-of-N refinement (Khairi et al., 2025). Mixture-of-Agents (Wang et al., 2024) layers multiple LLMs but uses a fixed model assignment rather than adaptive routing. $\mathbf{V}_1$ (Singh et al., 2026) demonstrates that RSA suffers from diversity collapse (monotonically declining pass@N) and proposes pairwise self-verification as an orthogonal remedy. SQUEEZE EVOLVE addresses the same bottleneck from a complementary angle: multi-model orchestration preserves diverse reasoning lineages, while confidence-based routing delegates easy aggregation groups to cheaper models.

Verification and confidence signals. External verification spans outcome reward models (Cobbe et al., 2021), process reward models (Lightman et al., 2023), and generative verifiers (Zhang et al., 2025), while self-verification can improve reasoning (Weng et al., 2023), though its benefits are situation-dependent (Lu et al., 2025). DeepConf (Fu et al., 2025) uses token-level confidence to filter low-quality reasoning traces, achieving large token savings. SQUEEZE EVOLVE uses the same class of model-intrinsic confidence signals not to filter or verify candidates, but as a routing signal that assigns each recombination group to a model, requiring no trained reward model or external evaluator.

LLM-driven evolutionary search. LLMs serve as evolutionary operators for discovering programs, prompts, and algorithms (Lehman et al., 2022; Romera-Paredes et al., 2024; Novikov et al., 2025; Yuksekgonul et al., 2026), with subsequent systems varying primarily in selection and variation strategies (Sharma, 2025; Lange et al., 2025; Agrawal et al., 2026; Assumpção et al., 2026; Cemri et al., 2026). EvoX (Liu et al., 2026) meta-evolves the search strategy itself rather than fixing it. These systems rely on external verifiers and apply a single model uniformly across all operators. SQUEEZE EVOLVE operates in the verifier-free regime and introduces adaptive model assignment: the evolutionary template remains unchanged, but each recombination group is routed to a model commensurate with its difficulty.

Model routing and cost-efficient inference. Cascading and routing frameworks route entire queries between a strong and a weak model (Ong et al., 2025; Valkanas et al., 2025). Arbitrage (Maheswaran et al., 2025) moves to finer granularity by routing individual reasoning steps between draft and target models, achieving $2\times$ latency reduction. SQUEEZE EVOLVE routes at a similarly fine granularity but within a multi-step evolutionary pipeline: individual recombination groups are assigned to models based on per-group confidence, and because these decisions compound across loops, savings accumulate multiplicatively.

10. Datasets and Benchmarks

Table 6 summarizes the benchmarks used in this work. We describe each below.

Table 6. Summary of evaluation benchmarks.

Benchmark	Size	Answer Format	Metric
AIME 2025	30	Integer (000–999)	Accuracy
HMMT Feb. 2025	30	Short answer	Accuracy
GPQA-Diamond	198	4-way MC	Accuracy
LiveCodeBench V6	175	Code	Pass@1
MMMU-Pro	1,730	Up to 10-way MC	Accuracy
BabyVision	388	Short answer	Accuracy
ARC-AGI-V2	120	Output grid	Pass@2
Circle Packing	1	Program	Objective

AIME 2025 (Balunović et al., 2026). The American Invitational Mathematics Examination consists of 30 problems (15 from AIME I, 15 from AIME II) covering algebra, geometry, number theory, and combinatorics. Each answer is an integer in $[0, 999]$, scored by exact match. We source problems via MathArena.

HMMT February 2025 (Balunović et al., 2026). The Harvard–MIT Mathematics Tournament February competition comprises 30 individual-round problems (10 Algebra, 10 Geometry, 10 Combinatorics). Answers are short numerical or symbolic expressions, scored by exact match. We source problems via MathArena.

GPQA-Diamond (Rein et al., 2024). A 198-question subset of GPQA filtered for maximum difficulty: both domain experts answered correctly while most non-experts failed even with unrestricted web access. Questions span graduate-level biology, physics, and chemistry in 4-way multiple-choice format.

LiveCodeBench V6 (Jain et al., 2024). A competitive programming benchmark sourcing problems from LeetCode, AtCoder, and Codeforces. Models generate code solutions evaluated against hidden test cases; we report pass@1. Continuous collection mitigates data contamination.

MMMU-Pro (Yue et al., 2025). A harder variant of MMMU spanning 1,730 multimodal questions across 30 subjects in six disciplines (Art & Design, Business, Science, Health & Medicine, Humanities & Social Science, Tech & Engineering). Answer choices are augmented from 4 to up to 10 options, and text-only solvable questions are filtered out.

BabyVision (Chen et al., 2026). A visual reasoning benchmark of 388 items across 22 subclasses in four categories: fine-grained discrimination, visual tracking, spatial perception, and visual pattern recognition. It tests core visual abilities independent of linguistic knowledge; human adults achieve 94.1% while leading MLLMs score below 50%. BabyVision uses an LLM-as-Judge (GPT-4o) for evaluation.

ARC-AGI-V2 (Chollet et al., 2025). A benchmark of 120 public evaluation tasks testing abstract reasoning and compositional generalization. Each task provides demonstration input–output grid pairs; the model must infer the transformation rule and produce the correct output grid. Scored by pass@2 across test pairs (exact grid match with two attempts).

Circle Packing (n=26) **(Novikov et al., 2025).** An open-ended optimization problem: pack 26 non-overlapping circles in a unit square to maximize the sum of their radii. This is a single continuous-objective instance used to evaluate evolutionary discovery capabilities. The metric is the objective value (sum of radii).

11. Generation Hyperparameters

Table 7 lists the generation hyperparameters used for each model. These are model-provided hyperparameters and differ from RSA except for GPT-OSS.

Table 7. Generation hyperparameters for each model.

Model	Effort	Temp.	Top-K	Top-P	Min-P	Gen. Len.
Qwen3-4B-Instruct-2507	—	0.7	20	0.8	0	8K
Qwen3-30B-A3B-Instruct-2507	—	0.7	20	0.8	0	8K
Qwen3-235B-A22B-Instruct-2507	—	0.7	20	0.8	0	16K
Qwen3-235B-A22B-Thinking-2507	—	0.6	20	0.95	0	32K
Qwen3-30B-A3B-Thinking-2507	—	0.6	20	0.95	0	32K
Qwen3-4B-Thinking-2507	—	0.6	20	0.95	0	32K
Qwen3.5-35B-A3B	—	1.0	20	0.95	0	64K
GPT-OSS-20B	medium	1	−1	1	0	16K
GPT-OSS-120B	medium	1	−1	1	0	16K
GPT-5 Mini	medium	default				32K
Gemini-3-Flash-Preview	high	default				64K
Gemini-3.1-Pro-Preview	high	default				64K
Kimi-2.5-Thinking	—	1.0	20	0.95	0	64K
Kimi-2.5-Instant	—	1.0	20	0.95	0	64K

12. Empirical Cost Model

We report per-token API pricing from commercial inference providers to ground the routing savings of SQUEEZE EVOLVE in real-world dollar costs. Table 8 lists the models used in our experiments together with their input and output prices from Alibaba Cloud, Together AI, Google, and OpenAI.

Table 8. Per-token API pricing ($/1M tokens) for each model used in our experiments.

Provider	Model	Input price	Output price
Alibaba Cloud	Qwen3-30B-A3B-Instruct-2507	$0.108	$0.431
	Qwen3-30B-A3B-Thinking-2507	$0.108	$1.076
	Qwen3-235B-A22B-Instruct-2507	$0.287	$0.920
	Qwen3.5-35B-A3B	$0.057	$0.459
Google Gemini API	Gemini-3-Flash-Preview	$0.50	$3.00
	Gemini-3.1-Pro-Preview	$2.00	$12.00
OpenAI	GPT-5 Mini	$0.25	$2.00
Together AI	GPT-OSS-20B	$0.05	$0.20
	GPT-OSS-120B	$0.15	$0.60
	Kimi-K2.5	$0.50	$2.80

13. Algorithm

Algorithm 1 SQUEEZE EVOLVE

Require: Query set $\{Q_q\}$, Model 2: M_2, Model 1 M_1, fitness f, operators Select, Route, LiteAgg, Update, population size N, group size K, groups per problem M, loops T
Ensure: Final populations $\{\mathcal{P}_q^{(T)}\}$
Loop 0 — Initialization (Model 2 only):
for each problem q **do**
  $\mathcal{P}_q^{(0)} \leftarrow \{\tau_i \sim p_{M_2}(\cdot \mid Q_q)\}_{i=1}^{N}$
end for
Loops 1…T — Fitness-routed evolution:
for $t = 1, \ldots, T$ **do**
  for each problem q **do**
    Score every $\tau \in \mathcal{P}_q^{(t-1)}$: compute $f(\tau)$
    $\mathcal{G}_q \leftarrow \mathrm{Select}(\mathcal{P}_q^{(t-1)},\ K,\ M,\ f)$
    $F(g) \leftarrow \mathrm{GroupFitness}(g, f)$ for each $g \in \mathcal{G}_q$
    $(\mathcal{B}_1,\ \mathcal{B}_2,\ \mathcal{B}_{\text{lite}}) \leftarrow \mathrm{Route}(\mathcal{G}_q,\ F)$
  end for
  $\mathcal{R}_1 \leftarrow \mathrm{Agg}(M_1, \mathcal{B}_1) \parallel \mathcal{R}_2 \leftarrow \mathrm{Agg}(M_2, \mathcal{B}_2) \parallel \mathcal{R}_{\text{lite}} \leftarrow \mathrm{LiteAgg}(\mathcal{B}_{\text{lite}})$
  $\mathcal{P}_q^{(t)} \leftarrow \mathrm{Update}\big(\mathcal{P}_q^{(t-1)},\ \mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_{\text{lite}}\big)$
end for

14. Full Aggregation Accuracy Results

Accuracy rises monotonically with the number of correct seeds. The large model maintains a consistent advantage at intermediate seed counts (1–3), while both models converge at the extremes (0 and 4).

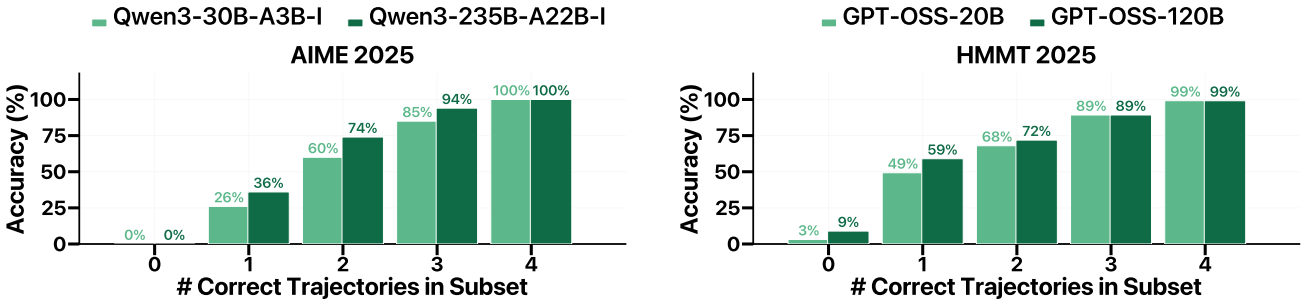


Figure 10. **Full aggregation accuracy vs. number of correct trajectories in subset.** *Left:* AIME 2025 (Qwen3-30B-A3B-Instruct vs. Qwen3-235B-A22B-Instruct). *Right:* HMMT 2025 (GPT-OSS-20B vs. GPT-OSS-120B).

15. Full Group Confidence Results

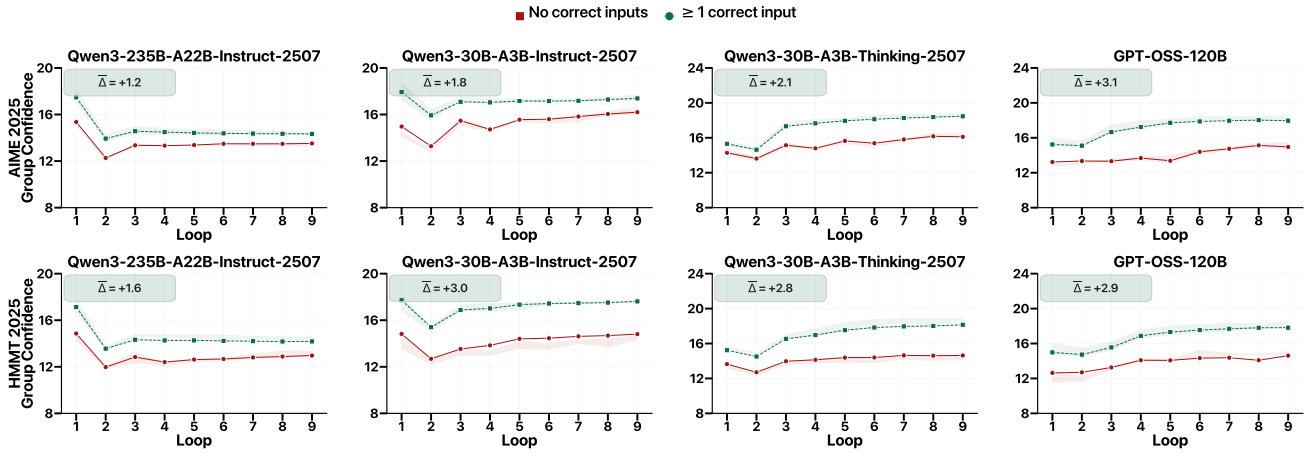


Figure 11. **Self-model group confidence by correctness across all baseline models.** Mean GC (with 40th–60th percentile band) across RSA loops 1–9 for four scorer models: Qwen3-235B-A22B-Instruct, Qwen3-30B-A3B-Instruct, Qwen3-30B-A3B-Thinking, and GPT-OSS-120B. Top row: AIME 2025; bottom row: HMMT 2025. Across all models and benchmarks, subsets containing correct trajectories maintain consistently higher GC than all-incorrect subsets.

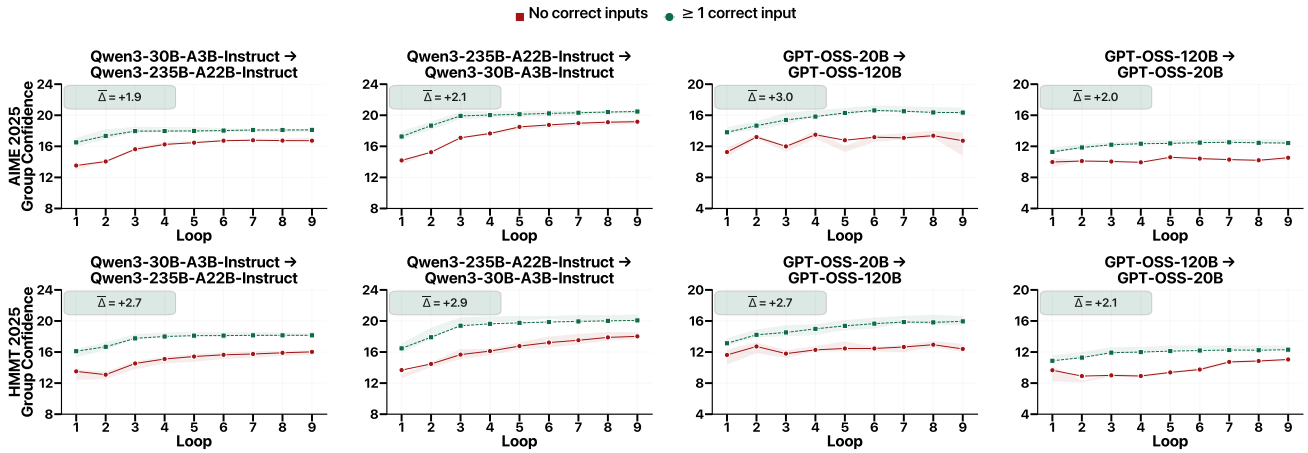


Figure 12. **Cross-model group confidence by correctness across all routing configurations.** Mean GC (with 40th–60th percentile band) for groups containing no correct inputs vs. groups with $\geq$1 correct input, pooled across seeds. Columns show four routing pairs: forward routing (Qwen3-30B-A3B-Instruct $\rightarrow$ Qwen3-235B-A22B-Instruct and GPT-OSS-20B $\rightarrow$ GPT-OSS-120B) and reverse routing (Qwen3-235B-A22B-Instruct $\rightarrow$ Qwen3-30B-A3B-Instruct and GPT-OSS-120B $\rightarrow$ GPT-OSS-20B). Top row: AIME 2025; bottom row: HMMT 2025. GC reliably separates correct from incorrect groups across all routing directions, model families, and benchmarks.

16. Empirical Cost Results: Homogeneous Model Pairs for Reasoning Tasks

Table 9. Empirical (dollar) cost results across datasets and model configurations. $/Prob is the average API cost per problem.

Data	Strategy	Model 1	Model 2	Acc.	$/Prob	$ Savings
AIME25	RSA	Qwen3-30B-A3B-I	—	77.8	$0.33	—
	RSA	—	Qwen3-30B-A3B-T	89.2	$0.94	1.0×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	Qwen3-30B-A3B-T	90.7	$0.66	1.4×
	RSA	Qwen3-30B-A3B-I	—	77.8	$0.33	—
	RSA	—	Qwen3-235B-A22B-I	82.0	$0.79	1.0×
	SQUEEZE EVOLVE (p=10)	Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	80.1	$0.47	1.7×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	81.0	$0.39	2.0×
	RSA	GPT-OSS-20B	—	90.0	$0.17	—
	RSA	—	GPT-OSS-120B	90.1	$0.34	1.0×
	SQUEEZE EVOLVE (p=10)	GPT-OSS-20B	GPT-OSS-120B	90.5	$0.21	1.6×
	SQUEEZE EVOLVE (p=0)	GPT-OSS-20B	GPT-OSS-120B	90.8	$0.18	1.9×
HMMT25	RSA	Qwen3-30B-A3B-I	—	57.7	$0.35	—
	RSA	—	Qwen3-30B-A3B-T	74.6	$1.10	1.0×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	Qwen3-30B-A3B-T	76.7	$0.77	1.4×
	RSA	Qwen3-30B-A3B-I	—	57.7	$0.35	—
	RSA	—	Qwen3-235B-A22B-I	72.1	$0.89	1.0×
	SQUEEZE EVOLVE (p=10)	Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	71.4	$0.52	1.7×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	67.4	$0.44	2.0×
	RSA	GPT-OSS-20B	—	80.8	$0.23	—
	RSA	—	GPT-OSS-120B	89.7	$0.41	1.0×
	SQUEEZE EVOLVE (p=10)	GPT-OSS-20B	GPT-OSS-120B	92.0	$0.25	1.6×
	SQUEEZE EVOLVE (p=0)	GPT-OSS-20B	GPT-OSS-120B	87.9	$0.22	1.8×
GPQA-Diamond	RSA	Qwen3-30B-A3B-I	—	72.5	$0.23	—
	RSA	—	Qwen3-30B-A3B-T	74.0	$0.57	1.0×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	Qwen3-30B-A3B-T	75.9	$0.32	1.8×
	RSA	Qwen3-30B-A3B-I	—	72.5	$0.23	—
	RSA	—	Qwen3-235B-A22B-I	84.3	$0.51	1.0×
	SQUEEZE EVOLVE (p=10)	Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	84.0	$0.30	1.7×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	83.8	$0.25	2.1×
	RSA	GPT-OSS-20B	—	75.0	$0.10	—
	RSA	—	GPT-OSS-120B	79.6	$0.16	1.0×
	SQUEEZE EVOLVE (p=10)	GPT-OSS-20B	GPT-OSS-120B	79.5	$0.10	1.6×
	SQUEEZE EVOLVE (p=0)	GPT-OSS-20B	GPT-OSS-120B	79.0	$0.09	1.9×
LCB-V6	RSA	Qwen3-30B-A3B-I	—	46.1	$0.19	—
	RSA	—	Qwen3-30B-A3B-T	64.2	$0.82	1.0×
	SQUEEZE EVOLVE (p=10)	Qwen3-30B-A3B-I	Qwen3-30B-A3B-T	62.7	$0.63	1.3×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	Qwen3-30B-A3B-T	59.1	$0.51	1.6×
	RSA	Qwen3-30B-A3B-I	—	46.1	$0.19	—
	RSA	—	Qwen3-235B-A22B-I	59.1	$0.33	1.0×
	SQUEEZE EVOLVE (p=10)	Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	55.9	$0.22	1.5×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	55.3	$0.19	1.7×
	RSA	GPT-OSS-20B	—	68.9	$0.14	—
	RSA	—	GPT-OSS-120B	75.9	$0.44	1.0×
	SQUEEZE EVOLVE (p=10)	GPT-OSS-20B	GPT-OSS-120B	75.6	$0.22	2.0×
	SQUEEZE EVOLVE (p=0)	GPT-OSS-20B	GPT-OSS-120B	73.3	$0.18	2.4×

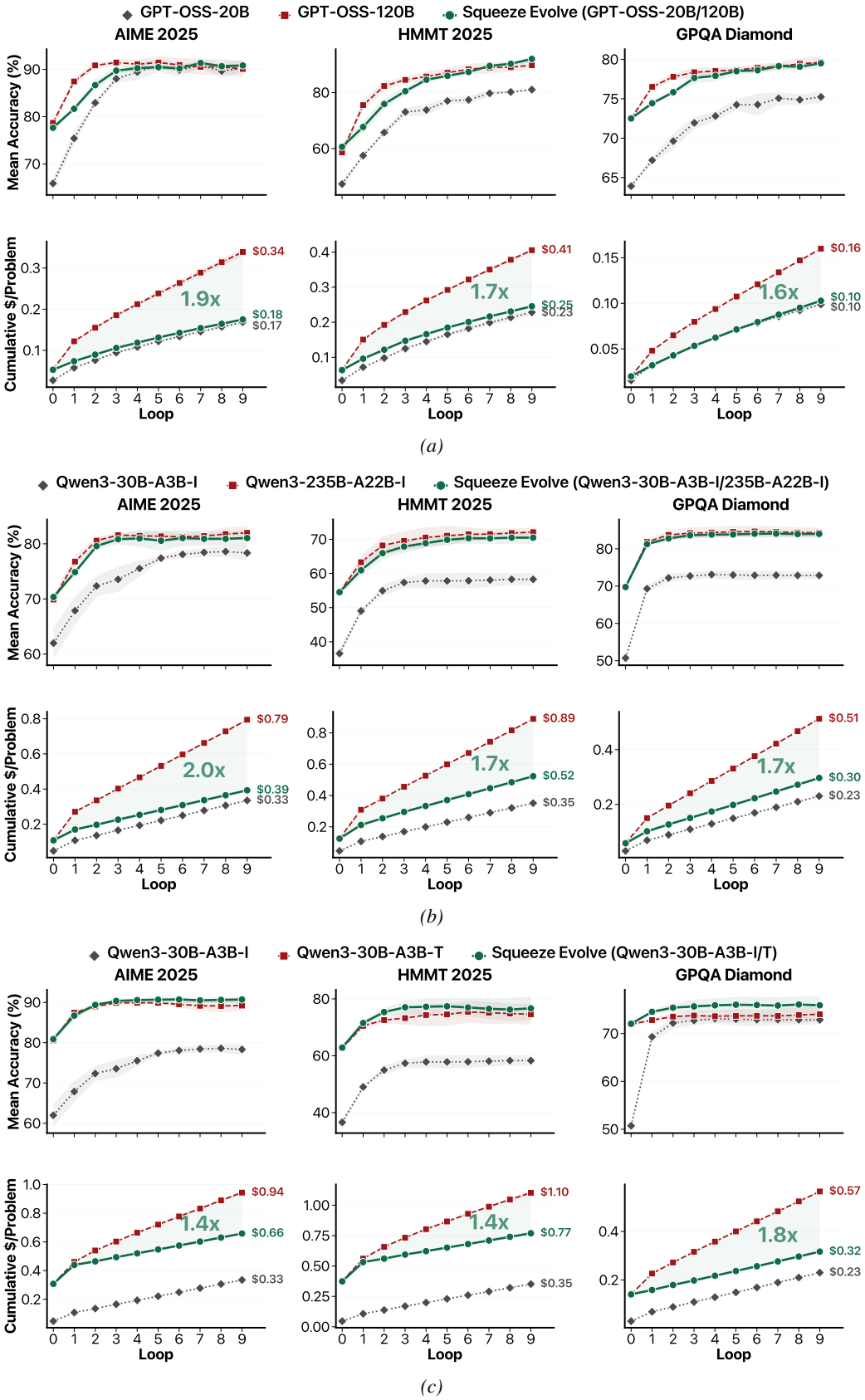
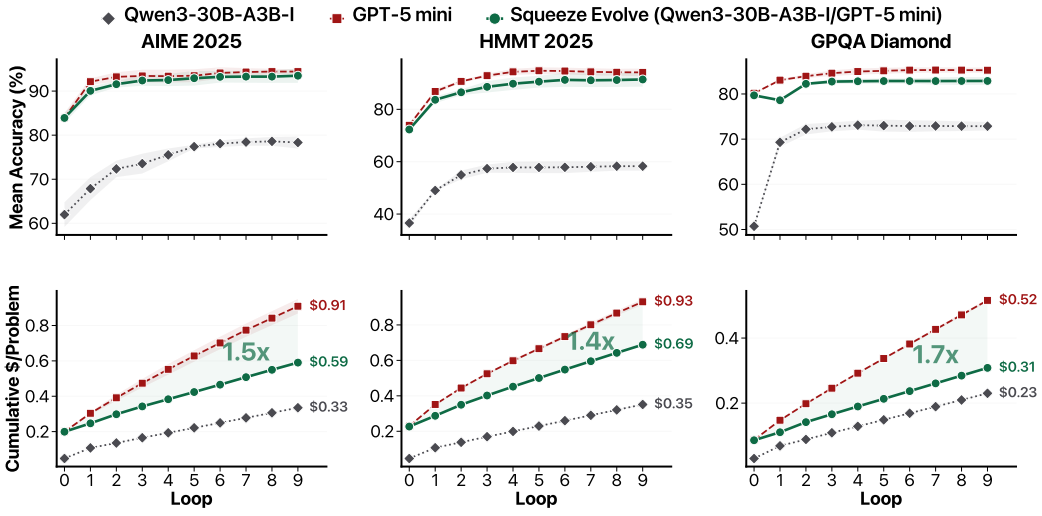
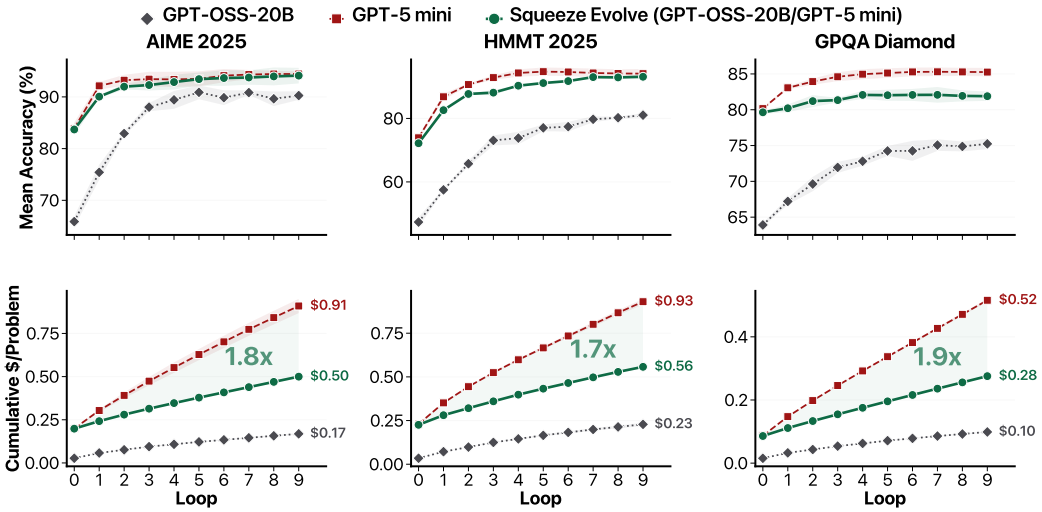


Figure 13. **Empirical cost results for homogeneous model pairs.** *Top row* of each panel: mean accuracy (%) across RSA loops. *Bottom row*: cumulative API cost per problem ($). SQUEEZE EVOLVE (green) matches or exceeds the Model 2-only baseline (red) in accuracy while substantially reducing cost. The shaded region highlights the cost savings, which grow with each loop as more candidates are routed to the cheaper Model 1.



(a)



(b)

Figure 14. **Empirical cost results for heterogeneous model pairs.** *Top row* of each panel: mean accuracy (%) across RSA loops. *Bottom row*: cumulative API cost per problem ($). SQUEEZE EVOLVE (green) matches or exceeds the Model 2 baseline (red) in accuracy while substantially reducing cost. The shaded region highlights the cost savings, which grow with each loop as more candidates are routed to the cheaper Model 1.

17. Empirical Cost Results: Heterogeneous Model Pairs for Reasoning Tasks

Table 10. Empirical (dollar) cost results across datasets and model configurations. $/Prob is the average API cost per problem.

Data	Strategy	Model 1	Model 2	Acc.	$/Prob	$ Savings
AIME25	RSA	Qwen3-30B-A3B-I	—	78.8	$0.34	—
	RSA	—	GPT-5 mini	94.2	$0.89	1.0×
	SQUEEZE EVOLVE (p=30)	Qwen3-30B-A3B-I	GPT-5 mini	93.5	$0.64	1.4×
	SQUEEZE EVOLVE (p=20)	Qwen3-30B-A3B-I	GPT-5 mini	93.5	$0.59	1.5×
	SQUEEZE EVOLVE (p=10)	Qwen3-30B-A3B-I	GPT-5 mini	93.1	$0.53	1.7×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	GPT-5 mini	91.9	$0.46	2.0×
	RSA	GPT-OSS-20B	—	90.6	$0.17	—
	RSA	—	GPT-5 mini	94.2	$0.89	1.0×
	SQUEEZE EVOLVE (p=30)	GPT-OSS-20B	GPT-5 mini	95.4	$0.50	1.8×

Continued on next page

Data	Strategy	Model 1	Model 2	Acc.	$/Prob	$ Savings
	SQUEEZE EVOLVE (p=20)	GPT-OSS-20B	GPT-5 mini	94.6	$0.46	1.9×
	SQUEEZE EVOLVE (p=10)	GPT-OSS-20B	GPT-5 mini	92.8	$0.39	2.3×
	SQUEEZE EVOLVE (p=0)	GPT-OSS-20B	GPT-5 mini	92.7	$0.30	3.0×
HMMT25	RSA	Qwen3-30B-A3B-I	—	58.9	$0.35	—
	RSA	—	GPT-5 mini	93.3	$0.94	1.0×
	SQUEEZE EVOLVE (p=30)	Qwen3-30B-A3B-I	GPT-5 mini	93.1	$0.69	1.4×
	SQUEEZE EVOLVE (p=20)	Qwen3-30B-A3B-I	GPT-5 mini	90.2	$0.66	1.4×
	SQUEEZE EVOLVE (p=10)	Qwen3-30B-A3B-I	GPT-5 mini	88.1	$0.59	1.6×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	GPT-5 mini	87.6	$0.51	1.9×
	RSA	GPT-OSS-20B	—	81.2	$0.22	—
	RSA	—	GPT-5 mini	93.3	$0.94	1.0×
	SQUEEZE EVOLVE (p=30)	GPT-OSS-20B	GPT-5 mini	93.1	$0.56	1.7×
	SQUEEZE EVOLVE (p=20)	GPT-OSS-20B	GPT-5 mini	91.8	$0.51	1.8×
	SQUEEZE EVOLVE (p=10)	GPT-OSS-20B	GPT-5 mini	89.8	$0.43	2.2×
	SQUEEZE EVOLVE (p=0)	GPT-OSS-20B	GPT-5 mini	89.3	$0.35	2.7×
GPQA-Diamond	RSA	Qwen3-30B-A3B-I	—	73.3	$0.23	—
	RSA	—	GPT-5 mini	85.0	$0.52	1.0×
	SQUEEZE EVOLVE (p=30)	Qwen3-30B-A3B-I	GPT-5 mini	82.6	$0.37	1.4×
	SQUEEZE EVOLVE (p=20)	Qwen3-30B-A3B-I	GPT-5 mini	83.6	$0.35	1.5×
	SQUEEZE EVOLVE (p=10)	Qwen3-30B-A3B-I	GPT-5 mini	83.2	$0.31	1.7×
	SQUEEZE EVOLVE (p=0)	Qwen3-30B-A3B-I	GPT-5 mini	82.2	$0.26	2.0×
	RSA	GPT-OSS-20B	—	75.5	$0.10	—
	RSA	—	GPT-5 mini	85.0	$0.52	1.0×
	SQUEEZE EVOLVE (p=30)	GPT-OSS-20B	GPT-5 mini	82.1	$0.27	1.9×
	SQUEEZE EVOLVE (p=20)	GPT-OSS-20B	GPT-5 mini	81.8	$0.25	2.1×
	SQUEEZE EVOLVE (p=10)	GPT-OSS-20B	GPT-5 mini	80.5	$0.20	2.5×
	SQUEEZE EVOLVE (p=0)	GPT-OSS-20B	GPT-5 mini	78.8	$0.16	3.3×

18. Empirical Cost Results: Homogeneous Model Pairs for Vision Tasks

Table 11. Empirical (dollar) cost results for vision tasks with homogeneous model pairs. $/Prob is the average API cost per problem.

Data	Strategy	Model 1	Model 2	Acc.	$/Prob	$ Savings
MMMU-Pro	RSA	Kimi-2.5-Instant	—	77.46	$0.29	—
	RSA	—	Kimi-2.5-Thinking	78.58	$1.04	1.0×
	SQUEEZE EVOLVE (p=0)	Kimi-2.5-Instant	Kimi-2.5-Thinking	78.63	$0.58	1.8×
BabyVision	RSA	Kimi-2.5-Instant	—	36.61	$0.29	—
	RSA	—	Kimi-2.5-Thinking	43.23	$2.05	1.0×
	SQUEEZE EVOLVE (p=0)	Kimi-2.5-Instant	Kimi-2.5-Thinking	41.56	$0.81	2.5×

19. Empirical Cost Results: Heterogeneous Model Pairs for Vision Tasks

Table 12. Empirical (dollar) cost results for vision tasks with heterogeneous model pairs. $/Prob is the average API cost per problem. †Image tokens are only processed by the Model 2 in loop 0; subsequent loops use Model 1 as a text-only causal LM (no image input).

Data	Strategy	Model 1	Model 2	Acc.	$/Prob	$ Savings
MMMU-Pro	RSA	Qwen3.5-35B-A3B†	—	—	—	—
	RSA	—	Kimi-2.5-Thinking	78.58	$1.04	1.0×
	SQUEEZE EVOLVE (p=0)	Qwen3.5-35B-A3B†	Kimi-2.5-Thinking	79.06	$0.46	2.3×
BabyVision	RSA	Qwen3.5-35B-A3B†	—	—	—	—
	RSA	—	Kimi-2.5-Thinking	43.23	$2.05	1.0×
	SQUEEZE EVOLVE (p=0)	Qwen3.5-35B-A3B†	Kimi-2.5-Thinking	41.27	$0.83	2.5×

20. Prefill Engine Microbenchmark

As described in Section 5.2, the SQUEEZE EVOLVE custom prefill engine computes the confidence scalar directly on GPU and returns only a single float per request, bypassing the native vLLM path that materializes full token-level logprob tensors and serializes them over HTTP. Table 13 reports per-request confidence-scoring latency for both paths across two models (GPT-OSS-120B and Qwen3-30B-A3B-Thinking-2507) and three context lengths (40K, 80K, and 120K tokens), measured at batch size 1 with 3 trials per configuration. The custom engine achieves 9.1–10.3× speedup on GPT-OSS-120B and 4.2–6.6× on Qwen3-30B-A3B-Thinking. The larger speedup on the 120B model reflects the heavier serialization and transfer burden at larger model scales: the native path transfers ∼13 MB of token-level logprob data per request regardless of model size, while the custom engine returns only ∼100 bytes. At the largest scale (Qwen3-235B-A22B), the native path OOMs entirely when materializing full prompt-logprob tensors, making the custom engine a prerequisite for confidence-based routing at this model size. Figure 15 shows that latency scales approximately linearly with context length under both paths, but the custom engine's slope is substantially gentler. This is because the native path's cost is dominated by tensor materialization and HTTP transfer, both of which grow with sequence length, whereas the custom engine performs an in-place reduction on GPU and transmits only the scalar result.

Table 13. **Prefill engine microbenchmark.** Confidence-scoring latency (seconds, mean over 3 trials) for native vLLM and vLLM SQUEEZE EVOLVE, measured at batch size 1 across three context lengths. vLLM SQUEEZE EVOLVE computes the confidence scalar on GPU and returns only a single float per request, reducing transfer volume from ∼13 MB to ∼100 bytes. Native vLLM OOMs on Qwen3-235B-A22B because materializing full prompt-logprob tensors at this model scale exceeds memory; vLLM SQUEEZE EVOLVE does not exhibit this issue.

Model	Context	vLLM native (s)	vLLM SQUEEZE EVOLVE (s)	Speedup
GPT-OSS-120B	40K	8.60	0.83	10.3×
	80K	17.68	1.79	9.9×
	120K	26.86	2.94	9.1×
Qwen3-30B-A3B	40K	10.34	1.58	6.6×
	80K	22.79	4.42	5.2×
	120K	35.42	8.41	4.2×

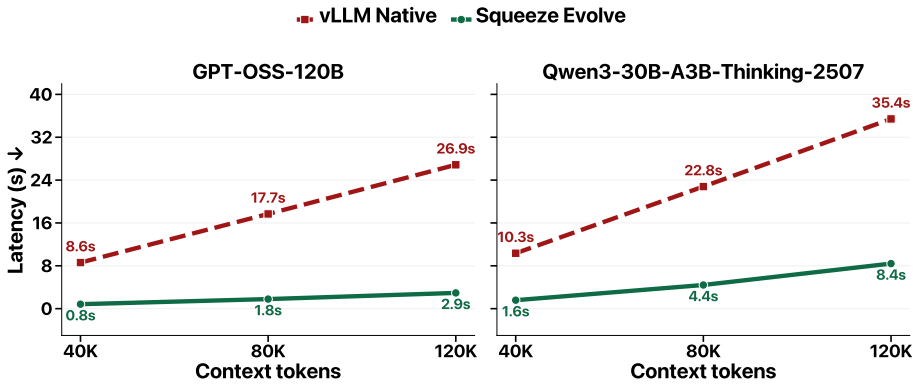


Figure 15. **Prefill engine speedup over native vLLM.** Confidence-scoring latency as a function of context length for GPT-OSS-120B (left) and Qwen3-30B-A3B-Thinking-2507 (right). The SQUEEZE EVOLVE custom prefill engine (green) computes the confidence scalar on GPU and returns only the final score, achieving 4–10× lower latency than the native vLLM prompt-logprob path (red), which materializes full token-level tensors and transfers them over HTTP. Speedup is larger for the 120B model where serialization and transfer costs dominate. Both paths scale approximately linearly with context length, but the custom engine has a substantially gentler slope.

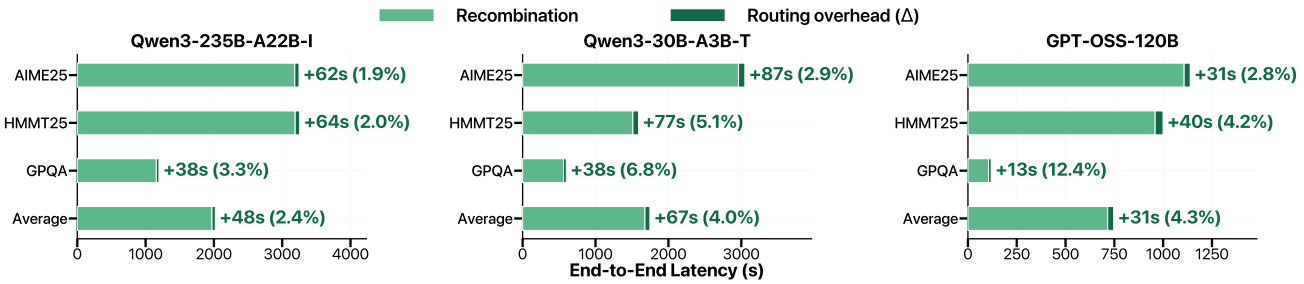


Figure 16. **Routing overhead is minimal.** Routing adds 1.9–6.8% to end-to-end latency for the Qwen models and 2.8–12.4% for GPT-OSS-120B, with the higher relative overhead on GPQA reflecting its short absolute generation time (106s). Full results in Table 14.

21. Routing Overhead Results

Experimental setup. For each benchmark and model M_2, we fix the full inference configuration: evolution parameters $(N = 16, K = 4, T = 10)$, prompts, decoding limits, hardware, serving engine, and batching policy. We compare two conditions. **RSA-M_2** is standard RSA with all calls executed by M_2 and no routing logic. **SQUEEZE EVOLVE-M_2** enables confidence scoring and threshold computation, but forces every aggregation call to M_2. This second condition preserves the routing machinery while removing any latency change due to sending work to M_1. The difference between RSA-M_2 and SQUEEZE EVOLVE-M_2 therefore isolates the routing overhead itself. This setup is also a conservative worst-case measurement: although SQUEEZE EVOLVE normally reduces latency by routing a subset of aggregations to M_1, here all aggregation work is still pinned to M_2.

Measurement protocol. We measure latency in a single-request setting, processing one problem at a time so that routing overhead is not confounded by cross-request queueing effects. Within each problem, however, we preserve the production execution strategy: the N confidence-scoring calls and aggregation calls for a loop are batched exactly as in normal serving. For each problem we log end-to-end latency, per-loop routing time, and per-loop aggregation time. We repeat the measurement across the evaluation set and report mean latency.

Overhead definition. Let T_{RSA} denote the end-to-end latency of RSA-M_2 and let T_{SQE} denote the latency of SQUEEZE EVOLVE-M_2. We define the absolute routing overhead as

$$\Delta_{\text{route}} = T_{\text{SQE}} - T_{\text{RSA}}.$$

We define the relative routing overhead as

$$\text{Overhead}(\%) = 100 \times \frac{T_{\text{SQE}} - T_{\text{RSA}}}{T_{\text{RSA}}}.$$

At the loop level, we decompose latency as

$$T_{\text{loop}} = T_{\text{routing}} + \sum_{i=1}^{N} T_{\text{agg}}(m_i),$$

where T_{routing} includes both prefill-only confidence scoring and percentile thresholding / dispatch, and $T_{\text{agg}}(m_i)$ is the aggregation time for the selected model $m_i \in \{M_1, M_2\}$. In this section, all aggregations are forced to M_2, so the observed gap isolates the latency overhead of routing logic alone.

Results. Table 14 reports per-benchmark routing overhead for each model M_2. Across all configurations, routing adds 1.9–6.8% to end-to-end latency for the Qwen models and 2.8–12.4% for GPT-OSS-120B, with cross-model averages of 2.4–4.3%. The higher relative overhead on GPQA-Diamond for GPT-OSS-120B (12.4%) reflects its short absolute generation time (106s), which makes the fixed routing cost proportionally larger.

Table 14. Routing-overhead measurements in the single-request setting. Comparing RSA-M_2 and SQUEEZE EVOLVE-M_2 isolates the latency overhead of confidence scoring and dispatch. Highlighted average rows summarize per-target-model averages across benchmarks.

M_L	Benchmark	RSA (s)	SQUEEZE EVOLVE (s)	Δ_{route} (s)	Overhead (%)
Qwen3-30B-A3B-T	AIME25	2961.72	3048.44	86.72	2.93%
	HMMT25	1512.39	1589.61	77.22	5.11%
	GPQA-Diamond	561.45	599.43	37.98	6.76%
	LCB-V6	1798.91	1864.46	65.55	3.64%
	Average	1678.52	1745.83	67.31	4.01%
Qwen3-235B-A22B-I	AIME25	3184.11	3246.08	61.97	1.95%
	HMMT25	3190.07	3253.75	63.68	2.00%
	GPQA-Diamond	1157.17	1195.57	38.40	3.32%
	LCB-V6	359.22	385.93	26.71	7.44%
	Average	1972.64	2020.33	47.69	2.42%
GPT-OSS-120B	AIME25	1107.44	1138.35	30.91	2.79%
	HMMT25	958.92	999.24	40.32	4.20%
	GPQA-Diamond	105.74	118.87	13.13	12.42%
	LCB-V6	691.80	729.70	37.90	5.47%
	Average	715.98	746.54	30.57	4.27%

22. System Throughput Results

Fairness rule. We compare RSA and SQUEEZE EVOLVE under the same total GPU budget G. RSA allocates all G GPUs to Model 2. SQUEEZE EVOLVE partitions the same budget into a Model 2 pool G_2 and a Model 1 pool G_1, subject to

$$G_2 + G_1 = G.$$

This fixed-budget constraint makes the throughput comparison deployment-fair.

Operating points. We sweep the routing percentile p from Section 5 across several values. Because the realized Model 1 routing share does not exactly equal $p/100$; we report both.

Pool sizing. Given a configured percentile p and its observed routing mix, we size the two pools so that their loop service times are approximately matched. Let $T_2(G_2)$ denote the time for the Model 2 pool to process the groups routed to Model 2, and let $T_1(G_1)$ denote the corresponding time for the Model 1 pool. We choose integer G_2 and G_1 satisfying $G_2 + G_1 = G$ and minimizing

$$|T_2(G_2) - T_1(G_1)|\,.$$

This latency-matching rule avoids provisioning a fast idle pool while the slower pool remains the throughput bottleneck.

Measurement protocol. After fixing the GPU split, we drive the system with enough concurrent requests to keep serving saturated and measure steady-state throughput. We report requests per second rather than tokens per second because Model 1 and Model 2 produce different numbers of output tokens for the same query, making a token-based metric an unfair comparison across routing configurations. Let N_{req} denote the total number of requests completed by the system over wall-clock interval Δt. We define throughput as

$$\text{Req/s} = \frac{N_{\text{req}}}{\Delta t}.$$

We use the same query stream, prompts, and serving engine for both RSA and SQUEEZE EVOLVE, and report completed requests per second after warmup. Because confidence-based routing causes Model 1 and Model 2 to observe different input and output lengths, we fix the input context length and output context length for each model to the values observed under the corresponding routing share.

Results. Table 15 reports steady-state throughput under a fixed total GPU budget G for two model pairs across four benchmarks. For each routing percentile p, the GPU budget is partitioned into large-model and small-model pools with approximately matched loop service times. The Accuracy column reports mean accuracy from Table 9.

Table 15. Fixed-budget system throughput under saturated serving. RSA and SQUEEZE EVOLVE use the same total GPU budget G. For each routing percentile p, SQUEEZE EVOLVE partitions the budget into Model 2 and Model 1 pools with approximately matched loop service times under the observed routing mix. Throughput is reported as steady-state completed requests per second after warmup. Accuracy is mean accuracy (%) from Table 9.

Model 1	Model 2	Benchmark	Strategy	Obs. M_1 share	GPU split (G_2:G_1)	Req/s	Speedup	Acc. (%)
Qwen3-30B-A3B-I	Qwen3-235B-A22B-I	AIME25	RSA	0%	16:0	1.36	1.00×	82.0
			SQUEEZE EVOLVE (p=10)	88.9%	8:8	7.41	5.44×	80.1
			SQUEEZE EVOLVE (p=0)	100%	0:16	13.47	9.90×	81.0
		HMMT25	RSA	0%	16:0	1.23	1.00×	72.1
			SQUEEZE EVOLVE (p=10)	87.5%	8:8	4.95	4.04×	71.4
			SQUEEZE EVOLVE (p=0)	100%	0:16	13.02	10.63×	67.4
		GPQA Diamond	RSA	0%	16:0	2.05	1.00×	84.3
			SQUEEZE EVOLVE (p=10)	87.5%	8:8	8.17	3.98×	84.0
			SQUEEZE EVOLVE (p=0)	100%	0:16	22.00	10.71×	83.8
		LCB-V6	RSA	0%	16:0	3.83	1.00×	59.1
			SQUEEZE EVOLVE (p=10)	87.5%	8:8	15.07	3.93×	55.9
			SQUEEZE EVOLVE (p=0)	100%	0:16	31.42	8.20×	55.3
GPT-OSS-20B	GPT-OSS-120B	AIME25	RSA	0%	20:0	17.09	1.00×	90.1
			SQUEEZE EVOLVE (p=10)	87.5%	4:16	24.59	1.44×	90.5
			SQUEEZE EVOLVE (p=0)	100%	0:20	39.43	2.31×	90.8
		HMMT25	RSA	0%	12:0	8.56	1.00×	89.7
			SQUEEZE EVOLVE (p=10)	87.5%	4:8	14.50	1.69×	92.0
			SQUEEZE EVOLVE (p=0)	100%	0:12	16.83	1.97×	87.9
		GPQA Diamond	RSA	0%	16:0	30.34	1.00×	79.6
			SQUEEZE EVOLVE (p=10)	87.5%	4:12	53.54	1.76×	79.5
			SQUEEZE EVOLVE (p=0)	100%	0:16	86.30	2.84×	79.0
		LCB-V6	RSA	0%	12:0	5.66	1.00×	75.9
			SQUEEZE EVOLVE (p=10)	87.1%	4:8	14.30	2.53×	75.6
			SQUEEZE EVOLVE (p=0)	100%	0:12	19.02	3.36×	73.3

23. ARC-AGI-V2 Complete Results

Table 16. ARC-AGI-V2 full results. Ancestor model: Gemini 3.1 Pro (High Thinking), N=4, K=2 T=10. †Uses code execution and program synthesis.

Strategy	Model 1	Model 2	Acc.	$/Task	Savings
Human baseline					
Human panel	—	—	100.0	$17.00	—
Single-shot baselines					
GPT-5.4 Pro (xhigh)	—	—	92.2	$17.60	—
Gemini 3.1 Pro (High)	—	—	88.1	$0.98	—
GPT-5.4 (xhigh)	—	—	84.2	$1.57	—
Claude Opus 4.6 (Thinking 120K, high)	—	—	79.0	$3.81	—
GPT-5.4 (high)	—	—	75.8	$1.08	—
Code-execution methods					
Imbue + Gemini 3.1 Pro†	—	—	95.1	$8.71	—
Confluence Lab†	—	—	97.9	$11.77	—
RSA	—	Gemini 3.0 Flash	45.0	$9.83	—
RSA	—	Gemini 3.1 Pro	93.3	$28.85	1.0×
SQUEEZE EVOLVE	—	Gemini 3.1 Pro	97.5	$7.74	3.7×

24. Circle Packing Complete Results

24.1. Algorithm summary

The evolved algorithm (Section 24.3) combines three strategies: (1) a diverse initialization ensemble that generates hundreds of candidate center layouts via hexagonal lattices, greedy farthest-point insertion, jittered grids, and random placements, scoring each with an exact linear programmed (LP) that maximizes $\sum r_i$ for fixed centres; (2) a hybrid optimization pipeline integrating LP-guided simulated annealing with SLSQP gradient-based refinement, where each stochastic perturbation of 1–3 centers is immediately followed by an LP solve to obtain provably optimal radii, and an adaptive temperature schedule prevents premature convergence before a final SLSQP pass jointly optimizes all $3N{=}78$ variables under wall-distance and non-overlap constraints; and (3) a principled decomposition that separates the hard combinatorial center placement ($\mathbb{R}^{52}$) from the easy convex radius assignment (an LP).

24.2. Hyperparameters

We instantiate SQUEEZE EVOLVE with GPT-OSS-120B and GPT-OSS-20B as M2 and M1 models, use group confidence as the fitness signal, fitness-weighted sampling ($\zeta{=}0.5$) for selection, a fixed confidence threshold at the 50th percentile for routing, and the *accumulate* update rule (Table 3), with $N{=}128$, $K{=}4$, $T{=}50$. At termination, we draw N candidates from the cumulative pool via confidence-weighted sampling and report the highest circle packing score.

24.3. Source code

```python
#!/usr/bin/env python3
"""
Optimised packing of 26 non-overlapping circles inside the unit square.

The algorithm combines:
  * Several diverse initialisation strategies (hexagonal lattice,
    farthest point, random grid, pure random).
  * A cheap but exact linear programme (LP) that, for any fixed set of
    centre positions, yields the maximal radii (maximising the sum of radii).
  * Stochastic hill-climbing with a temperature schedule to refine the
    centre positions.
  * A final local optimisation with SciPy's SLSQP optimiser, which moves
    centres and radii simultaneously while respecting all constraints.
  * A tiny post-processing step that enforces strict feasibility.

The program prints exactly 26 lines "x y r" (nine decimal digits each)
to stdout.
"""

import sys
import numpy as np

try:
    from scipy.optimize import linprog, minimize
    _SCIPY = True
except Exception:
    _SCIPY = False

N = 26
PAIR_I, PAIR_J = np.triu_indices(N, 1)
M_PAIRS = len(PAIR_I)

# Pre-computed constraint matrix for the LP:
# one row per pair (r_i + r_j <= d_ij)
A_UB = np.zeros((M_PAIRS, N), dtype=float)
A_UB[np.arange(M_PAIRS), PAIR_I] = 1.0
A_UB[np.arange(M_PAIRS), PAIR_J] = 1.0

def _fallback_radii(centres: np.ndarray) -> np.ndarray:
    """Simple pairwise scaling - used only if the LP fails."""
    n = centres.shape[0]
    r = np.minimum.reduce([centres[:, 0],
                           centres[:, 1],
                           1.0 - centres[:, 0],
                           1.0 - centres[:, 1]])
    for i in range(n):
        for j in range(i + 1, n):
            d = np.linalg.norm(centres[i] - centres[j])
            if r[i] + r[j] > d:
                scale = d / (r[i] + r[j])
                r[i] *= scale
                r[j] *= scale
    return np.maximum(r, 0.0)
```

```python


def _lp_optimal(centres: np.ndarray) -> tuple[np.ndarray, float]:
    """Solve the LP that maximises sum(r_i) for the fixed centres."""
    wall = np.minimum.reduce([centres[:, 0],
                              centres[:, 1],
                              1.0 - centres[:, 0],
                              1.0 - centres[:, 1]])
    bounds = [(0.0, w) for w in wall]

    diffs = centres[PAIR_I] - centres[PAIR_J]
    pair_dist = np.linalg.norm(diffs, axis=1)

    if _SCIPY:
        res = linprog(c=-np.ones(N),
                      A_ub=A_UB,
                      b_ub=pair_dist,
                      bounds=bounds,
                      method='highs',
                      options={'presolve': True})
        if res.success:
            radii = np.clip(res.x, 0.0, wall)
            return radii, float(radii.sum())
    radii = _fallback_radii(centres)
    return radii, float(radii.sum())

def _hex_lattice(dx: float) -> np.ndarray:
    """Generate points on a hexagonal lattice (up to 26 points)."""
    dy = dx * np.sqrt(3.0) / 2.0
    pts = []
    row = 0
    y = 0.0
    while y <= 1.0 + 1e-12:
        offset = 0.0 if (row % 2 == 0) else dx / 2.0
        x = offset
        while x <= 1.0 + 1e-12:
            pts.append([x, y])
            x += dx
        y += dy
        row += 1
    pts = np.asarray(pts)
    pts = pts[(pts[:, 0] >= 0.0) & (pts[:, 0] <= 1.0) &
              (pts[:, 1] >= 0.0) & (pts[:, 1] <= 1.0)]
    if pts.shape[0] > N:
        margins = np.minimum.reduce([pts[:, 0],
                                     pts[:, 1],
                                     1.0 - pts[:, 0],
                                     1.0 - pts[:, 1]])
        keep = np.argsort(-margins)[:N]
        pts = pts[keep]
    elif pts.shape[0] < N:
        rng = np.random.default_rng(0)
        extra = rng.uniform(0.0, 1.0, size=(N - pts.shape[0], 2))
        pts = np.vstack([pts, extra])
    return pts


def _init_farthest(rng: np.random.Generator, n: int = N) -> np.ndarray:
    """Greedy farthest-point placement (points stay inside [0,1]^2)."""
    pts = []
    pts.append(rng.uniform(0.1, 0.9, size=2))
    while len(pts) < n:
        cand = rng.uniform(0.1, 0.9, size=(200, 2))
        existing = np.asarray(pts)
        dists = np.linalg.norm(
            cand[:, None, :] - existing[None, :, :], axis=2)
        min_d = dists.min(axis=1)
        best = np.argmax(min_d)
        pts.append(cand[best])
    return np.asarray(pts)


def _init_grid_jitter(rng: np.random.Generator,
                      jitter: float = 0.02) -> np.ndarray:
    """5x5 grid (0.1-0.9) with small random jitter + one extra point."""
    xs = np.linspace(0.1, 0.9, 5)
    ys = np.linspace(0.1, 0.9, 5)
    xv, yv = np.meshgrid(xs, ys)
    base = np.column_stack([xv.ravel(), yv.ravel()])  # 25 points
    base += rng.uniform(-jitter, jitter, base.shape)
    base = np.clip(base, 0.01, 0.99)
```

```python
    extra = rng.uniform(0.01, 0.99, size=(1, 2))
    return np.vstack([base, extra])


def _init_random(rng: np.random.Generator) -> np.ndarray:
    """Pure uniform random points."""
    return rng.uniform(0.0, 1.0, size=(N, 2))

def _hill_climb(start: np.ndarray, rng: np.random.Generator,
                iterations: int = 1500
                ) -> tuple[np.ndarray, np.ndarray, float]:
    best_c = start.copy()
    best_r, best_sum = _lp_optimal(best_c)

    temperature = 0.02
    for it in range(iterations):
        cand_c = best_c.copy()
        k = rng.integers(1, 4)
        sel = rng.choice(N, size=k, replace=False)
        max_step = 0.09 * (1.0 - it / iterations) + 0.005
        cand_c[sel] += rng.normal(scale=max_step, size=(k, 2))
        cand_c = np.clip(cand_c, 0.0, 1.0)

        cand_r, cand_sum = _lp_optimal(cand_c)
        delta = cand_sum - best_sum

        if delta > 1e-9:
            best_c, best_r, best_sum = cand_c, cand_r, cand_sum
            temperature = max(temperature * 0.95, 1e-6)
        else:
            if (temperature > 0.0
                    and rng.random() < np.exp(delta / temperature)):
                best_c, best_r, best_sum = cand_c, cand_r, cand_sum
            temperature *= 0.9995
    return best_c, best_r, best_sum

def _refine_slsqp(centres: np.ndarray, radii_start: np.ndarray,
                  rng: np.random.Generator
                  ) -> tuple[np.ndarray, np.ndarray, float]:
    if not _SCIPY:
        return centres, radii_start, radii_start.sum()

    n = N
    x0 = np.empty(3 * n)
    x0[:n] = centres[:, 0]
    x0[n:2 * n] = centres[:, 1]
    x0[2 * n:] = radii_start

    bounds = ([(0.0, 1.0)] * n
              + [(0.0, 1.0)] * n
              + [(0.0, None)] * n)

    def cons_fun(x):
        xs = x[:n]
        ys = x[n:2 * n]
        rs = x[2 * n:]
        c1 = xs - rs
        c2 = ys - rs
        c3 = (1.0 - xs) - rs
        c4 = (1.0 - ys) - rs
        dx = xs[:, None] - xs[None, :]
        dy = ys[:, None] - ys[None, :]
        d = np.sqrt(dx * dx + dy * dy)
        iu = np.triu_indices(n, 1)
        c5 = d[iu] - (rs[iu[0]] + rs[iu[1]])
        return np.concatenate([c1, c2, c3, c4, c5])

    constraints = {'type': 'ineq', 'fun': cons_fun}

    def obj_fun(x):
        return -np.sum(x[2 * n:])

    res = minimize(obj_fun,
                   x0,
                   method='SLSQP',
                   bounds=bounds,
                   constraints=[constraints],
                   options={'ftol': 1e-9, 'maxiter': 2000,
                            'disp': False})

    if not res.success:
```

```python
        final_c = centres
    else:
        xs_opt = res.x[:n]
        ys_opt = res.x[n:2 * n]
        final_c = np.vstack([xs_opt, ys_opt]).T

    final_r, final_sum = _lp_optimal(final_c)
    return final_c, final_r, final_sum

def _make_feasible(centres: np.ndarray, radii: np.ndarray,
                   eps: float = 1e-12) -> np.ndarray:
    wall = np.minimum.reduce([centres[:, 0],
                              centres[:, 1],
                              1.0 - centres[:, 0],
                              1.0 - centres[:, 1]])
    radii = np.minimum(radii, wall - eps)
    for _ in range(5):
        changed = False
        for i in range(N):
            for j in range(i + 1, N):
                d = np.linalg.norm(centres[i] - centres[j])
                if radii[i] + radii[j] > d - eps:
                    scale = (d - eps) / (radii[i] + radii[j])
                    radii[i] *= scale
                    radii[j] *= scale
                    changed = True
        if not changed:
            break
    return np.maximum(radii, 0.0)

def construct_packing() -> tuple[np.ndarray, np.ndarray, float]:
    rng = np.random.default_rng(123456789)
    best_sum = -np.inf
    best_centres = None
    best_radii = None

    # -- Stage 1: diverse starts --
    candidates = []
    for dx in np.linspace(0.16, 0.30, 15):
        base = _hex_lattice(dx)
        for jitter_scale in (0.0, 0.008, 0.02, 0.04):
            for _ in range(5):
                jitter = rng.normal(
                    scale=jitter_scale, size=base.shape)
                centres = np.clip(base + jitter, 0.0, 1.0)
                radii, s = _lp_optimal(centres)
                candidates.append((s, centres, radii))

    for _ in range(5):
        centres = _init_farthest(rng)
        radii, s = _lp_optimal(centres)
        candidates.append((s, centres, radii))

    for jit in (0.015, 0.03):
        for _ in range(5):
            centres = _init_grid_jitter(rng, jitter=jit)
            radii, s = _lp_optimal(centres)
            candidates.append((s, centres, radii))

    for _ in range(5):
        centres = _init_random(rng)
        radii, s = _lp_optimal(centres)
        candidates.append((s, centres, radii))

    candidates.sort(key=lambda x: x[0], reverse=True)
    top_candidates = candidates[:6]

    # -- Stage 2: hill climbing --
    for s0, cent0, rad0 in top_candidates:
        cent_opt, rad_opt, sum_opt = _hill_climb(
            cent0, rng, iterations=1800)
        if sum_opt > best_sum:
            best_sum = sum_opt
            best_centres = cent_opt
            best_radii = rad_opt

    # -- Stage 3: SLSQP refinement --
    if best_centres is not None:
        refined_c, refined_r, refined_sum = _refine_slsqp(
            best_centres, best_radii, rng)
        if refined_sum > best_sum:
```

```python
            best_sum = refined_sum
            best_centres = refined_c
            best_radii = refined_r

    final_radii = _make_feasible(best_centres, best_radii)
    final_sum = float(final_radii.sum())
    return best_centres, final_radii, final_sum

def run_packing() -> None:
    centres, radii, _ = construct_packing()
    out = sys.stdout
    fmt = "{:.9f} {:.9f} {:.9f}\n"
    for i in range(N):
        out.write(fmt.format(
            centres[i, 0], centres[i, 1], radii[i]))


if __name__ == "__main__":
    run_packing()
```

Listing 1. Evolved circle packing program (n=26).

24.4. Correlation between confidence and score across loops

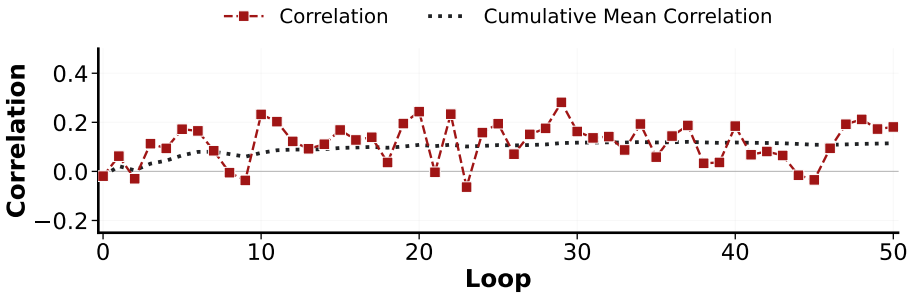


Figure 17. **Spearman rank correlation between confidence and score**