FlashEvolve: Accelerating Agent Evolution with Asynchronous Stage Orchestration

Zhengding Hu[1], Mingge Lu[1], Zhen Wang[1], Jixuan Ruan[1], Chang Chen[1], Zaifeng Pan[1], Yue Guan[1], Ruiyi Wang[1], Zhongkai Yu[1], Chao Zhang[2], Yufei Ding[1]
[1]University of California, San Diego [2]Georgia Institute of Technology

Abstract

LLM-based evolution has emerged as a promising way to improve agents by refining non-parametric artifacts, but its wall-clock cost remains a major bottleneck. We identify that this cost comes from synchronized stage execution and imbalance inside each LLM-heavy stage. We present FlashEvolve, an efficient framework that replaces synchronized execution with asynchronous workers and queues, allowing different stages and steps to overlap. To handle data staleness introduced by asynchrony, FlashEvolve tracks artifact versions and applies different policies to update, discard, or patch stale artifacts. Unlike weight-space staleness in asynchronous RL, language-space staleness is inspectable and repairable: a stale artifact is not just delayed work, but readable evidence that the LLM can reflect on, revise, and turn into useful evolution signal. FlashEvolve further improves throughput and token efficiency with speculative stage completion and adaptive workflow control.

On GEPA workloads, FlashEvolve improves proposal throughput by $3.5\times$ on local vLLM and $4.9\times$ on API serving over synchronous GEPA. The same design also applies to ACE and Meta-Harness. Our code repository is available at `https://github.com/Leo9660/FlashEvolve.git`.

1 Introduction

A growing line of recent work enables LLM agents to evolve themselves. Instead of updating model weights, these systems iteratively refine the non-parametric components that govern their behavior, including system prompts [2, 29, 30], context and memory [35, 22, 34], harness code [16, 14] and generated programs [20, 13, 3]. This emerging paradigm of test-time self-evolution [6] fundamentally relaxes the access requirements of weight-space adaptation: it requires neither the labeled trajectories used by supervised fine-tuning nor the gradient updates required by reinforcement learning. By having an LLM reflect on full execution traces rather than optimize against scalar rewards, this paradigm draws a richer learning signal from each rollout: GEPA [2] outperforms GRPO with an average gain of 6% across six reasoning benchmarks, while Meta-Harness [14] automatically discovers agent harnesses that surpass the best hand-engineered baselines on different domain-specific benchmarks.

Despite its algorithmic appeal, agent evolution remains expensive in wall-clock execution time. Existing evolution algorithms pursue "faster" evolution by improving the quality of each step through stronger reflection [2, 35, 33], better artifact proposal and search [13, 18], or larger-batch updates [15], thereby reducing the number of steps needed. However, fewer evolution steps do not necessarily translate into shorter wall-clock time. For example, on IFBench, a single GEPA evolution step already takes $\sim$1 minute; Combee [15] parallelizes proposal generation, but further stretches each step to $\sim$2.8 minutes. Reaching a stable improvement requires more than 2 hours on an H100 GPU. This cost further grows with data scale, making evolution runs slow to tune and deploy in practice.

Such high wall-clock cost comes from **synchronized stage execution**. As shown in Figure 1, each evolution step runs a sequence of LLM-heavy stages, such as running the current artifact on a mini-

40th Conference on Neural Information Processing Systems (NeurIPS 2026).

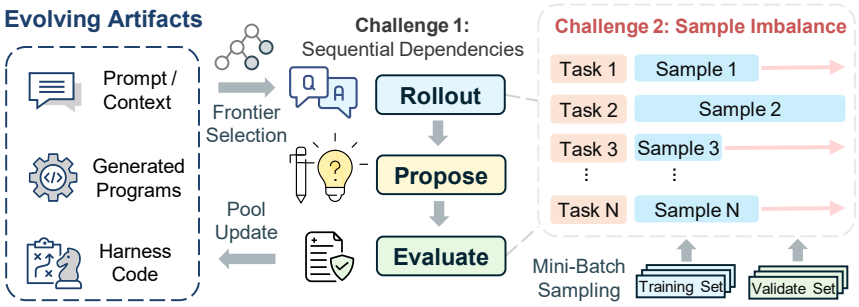


Figure 1: Illustration of the multi-stage execution in agent evolution. The synchronized stage orchestration in existing implementations [2, 35, 15] exposes two efficiency challenges: sequential dependencies across stages and sample workload imbalance within each individual stage.

batch of inputs, proposing a new candidate artifact, and evaluating the new one. A later stage cannot start until the previous stage has fully completed. Such serial structure prevents overlap across stages.

The cost inefficiency is amplified by **generation imbalance** inside each stage. Request lengths vary widely across samples, such as different validation samples in the evaluate stage. This creates a long-tail effect: the longest requests determine the execution time of the whole stage. This reduces the effective batch size in local inference frameworks [12, 38], agentic serving systems [23] and API-based remote calls, leading to low resource utilization and inefficient waiting for long samples.

To this end, we present FlashEvolve, a framework that improves the time efficiency of agent evolution through asynchronous stage orchestration. FlashEvolve treats an evolution loop as a set of LLM-heavy stages connected by queues. This allows artifact execution, proposal generation, evaluation, and pool update to overlap in time, turning a synchronized loop into a streaming execution pipeline.

This design introduces new systems challenges. Asynchronous execution can generate stale items because an artifact pool may change while earlier items are still waiting in queues. FlashEvolve handles this with artifact-version tracking and staleness-aware policies, including version comparison and discarding, or reflective patching for stale language artifacts. This property is specific to agent evolution. Unlike weight updates in SFT or reinforcement learning, evolution artifacts are prompts, memories, harness code, or programs. A stale artifact is therefore still an inspectable object: its relation to the current pool can be judged as complementary, redundant, or conflicting, and can be revised by the same LLM mechanism used for proposal. This makes staleness a semantic repair problem rather than only a scheduling hazard. FlashEvolve further reduces waiting inside long stages through speculative completion, and uses adaptive workflow control to balance workload across stages. Together, these mechanisms improve throughput while preserving the quality of evolution.

2 Background and Motivation

2.1 Agent Evolution: Self-Improvement Beyond Weight Updates

Agent evolution has emerged as a new paradigm for adapting LLM-based systems to new data and tasks [6, 4]. This success stems from the already strong reasoning capability of modern LLMs [7, 10], which enables a single model to reflect on its own trajectories [27], critique its own outputs [19], and propose new artifacts that govern its own behavior, ranging from prompts, memory, and harness code that govern how the agent operates, to generated programs that constitute the task solution. Crucially, this happens without modifying model weights, sidestepping the training infrastructure of supervised fine-tuning [28, 37] and reinforcement learning [7, 21] while delivering comparable or stronger gains.

For example, GEPA [2] and ACE [35] use reflection on execution traces to evolve system prompts and contextual playbooks. Meta-Harness [14] and AutoHarness [16] use a coding agent to evolve the harness based on prior runs and their failure modes. AlphaEvolve [20] and ShinkaEvolve [13] push this beyond the agent itself, evolving the generated programs the agent uses to solve problems, where the LLM acts as a mutation operator and an external evaluator scores each candidate.

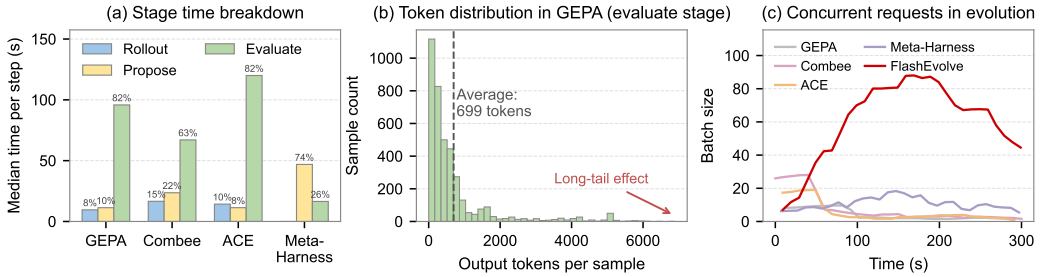


Figure 2: Profiling results of inefficiency in synchronized agent evolution. (a) Stage execution is serial, and stage time is highly imbalanced. (b) Within a single stage, output lengths show a long-tail distribution, so the slowest requests determine stage completion time. (c) Serial stage execution and intra-stage imbalance reduce effective concurrency, while FlashEvolve keeps more requests in flight.

An agent evolution loop iterates over multiple iteration steps, where each step consists of several stages, as illustrated in Figure 1. The LLM-heavy stages are typically *Generate*, *Propose*, and *Evaluate*. The Generate stage runs the current artifact on tasks to collect trajectories. The Propose stage reflects on these trajectories to produce a new candidate artifact. The Evaluate stage scores the candidate against task signals and filters out underperforming ones. A subsequent update commits the new artifact to the artifact pool. At the start of each step, new candidate artifacts are selected from the pool, through methods like Pareto-aware sampling [2] or evolutionary tournaments [20].

2.2 Inefficiency in Agent Evolution: Sequential and Imbalanced Stages

Despite its algorithmic appeal, agent evolution remains expensive in wall-clock time. Based on our experiments, even with state-of-the-art LLM serving infrastructure such as vLLM [12], which supports continuous batching and prefix caching, GEPA with Qwen3-8B takes 50 minutes to complete 49 evolution steps on IFBench [24], and 134 minutes to complete 411 steps on HotpotQA [32].

This inefficiency stems from sequential and synchronized stage execution. Each evolution step runs its LLM-heavy stages serially, and each stage internally waits for all parallel LLM requests to finish before advancing to the next stage. This structure produces two compounding costs. First, the serial chain forces total step time to be the sum of per-stage durations, with no opportunity to overlap stages. As shown in Figure 2(a), stage time is highly imbalanced, so different stages can become the bottleneck depending on the workload and algorithm. Second, the synchronization barrier at each stage's end forces the entire batch to wait for the slowest one. As shown in Figure 2(b), output lengths within a stage show a long-tail distribution, so a small number of long requests determine stage completion time. Consequently, sequential execution and intra-stage imbalance reduce effective concurrency and leave the LLM backend underutilized, as shown in Figure 2(c).

Such inefficiency cannot be solved by simply launching more LLM requests in parallel. Agent evolution must convert a synchronized multi-stage loop into a streaming workflow while preserving artifact-evolution semantics. This creates two challenges. First, asynchrony introduces artifact-level staleness: intermediate results may be produced from an artifact pool that has already changed before they are consumed. Second, naive parallel scaling can amplify workload imbalance: fast stages may overproduce items for slow stages, while long-tail requests within a stage can still delay downstream execution. This causes queue buildup, longer staleness windows, and wasted LLM work. These challenges require orchestration mechanisms that jointly manage staleness and workload balance.

Analogy to Asynchronous RL. These challenges are related to synchronous LLM RL systems [26, 1, 8], which also suffer from synchronization overhead and workload imbalance. Asynchronous RL addresses this by overlapping rollout generation with training and controlling off-policy optimization [5, 39, 25]. Agent evolution differs in two key ways. First, it contains multiple LLM inference stages rather than a single "rollout" stage in RL. Each stage has batched generation behavior and its own long-tail imbalance. Second, staleness occurs over inspectable language artifacts, such as prompts, memories, harness code, and programs, rather than continuous model weights. This allows a more flexible design space for staleness handling policies.

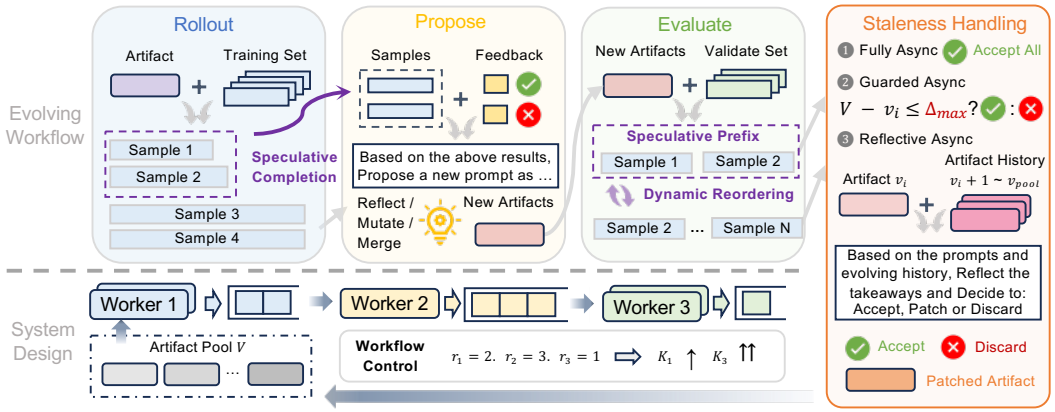


Figure 3: Overview of FlashEvolve. FlashEvolve executes agent evolution with asynchronous workers and queues across stages. Workers pass partial or completed results through queues instead of waiting for a whole stage to finish. FlashEvolve further uses speculative completion, validation-set reordering, workflow control, and staleness-aware handling to improve throughput while limiting data staleness.

3 FlashEvolve: Asynchronous Framework for Agent Evolution

We present FlashEvolve, an asynchronous framework that removes the sequential and imbalanced behavior identified in Section 2.2. FlashEvolve decomposes an evolution loop into asynchronous workers connected by queues, so different stages and evolution steps can overlap. Each queue item carries the artifact state and pool version, allowing FlashEvolve to detect stale items. On top of this execution model, FlashEvolve introduces staleness-aware data handling, speculative stage completion, and adaptive workflow control to improve the time efficiency of evolution.

3.1 Asynchronous Execution with Workers and Queues

Asynchronous workers. FlashEvolve turns a synchronized evolution step into asynchronous workers connected by queues. Instead of waiting for artifact proposal, validation, and pool update to finish before starting the next step, workers continuously process ready items and pass their outputs to downstream queues. This allows different stages and evolution steps to overlap. Each stage has an input queue and a set of workers. A queue item carries the artifact being evolved, the input/output, and the artifact-pool version v_i at item creation. The pool version increases after each pool update, so FlashEvolve can compare v_i with the current version v to detect stale items.

Worker concurrency. To improve system throughput, FlashEvolve assigns a worker count K_i to each asynchronous stage i. A larger K_i allows more tasks in stage i to issue LLM requests at the same time, which increases per-stage concurrency so the whole pipeline is not bottlenecked by the throughput of a single slow or imbalanced stage. The tradeoff is data staleness: larger worker counts increase the chance that queued items were generated from an older artifact pool state.

3.2 Staleness-Aware Data Handling

FlashEvolve supports three policies for handling such stale items with different tradeoffs:

- *Full Async* does not check artifact pool versions and allows all items to continue through the pipeline. This policy preserves all completed work and maximizes throughput, but stale items may introduce outdated updates into the artifact pool and impact convergence.
- *Guarded Async* discards an item when its version gap exceeds a threshold $\Delta_{\max}$. Let v_i denote the artifact-pool version used to generate item i, and let v denote the current artifact-pool version. The version gap is defined as $\Delta_i = v - v_i$. Guarded Async allows item i to continue only when $\Delta_i \leq \Delta_{\max}$; otherwise, it discards the item. This policy prevents highly stale items, but will waste the generated tokens that already spent on discarded items.

- *Reflective Async* inspects and updates stale items by adding a new reflection worker stage. For an item i with version gap $\Delta_i > 0$, the reflection worker uses the stale item and all artifact-pool updates between versions v_i and v to decide whether the item still contributes a useful change. If so, it patches the item against the current artifact pool state and lets it continue; otherwise, FlashEvolve discards it. Non-stale items continue without reflection. This policy avoids uncontrolled stale updates while reusing useful stale items, reducing wasted LLM generations.

Why language-space staleness can be repaired. Language-space staleness is discrete and inspectable, unlike parameter staleness in asynchronous RL, which is continuous and opaque. In RL, a stale item is tied to an older point in weight space, so systems typically handle it through importance weighting, bounded delay, or discard. In agent evolution, a stale item is text or code, such as a prompt edit, memory update, harness mutation, or generated program. FlashEvolve can therefore inspect the stale item together with the intervening artifact history and decide whether the edit is orthogonal, already subsumed, or conflicting with the current artifact pool. This makes repair a first-class operation: stale items can be patched when they contain reusable information, or discarded when they are too specific or inconsistent. Figure 5 shows an example where FlashEvolve filters task-specific stale content and keeps transferable principles to form a compact prompt patch.

3.3 Speculative Stage Completion

Asynchronous workers remove waiting between stages, but each worker may still wait for all LLM requests in its current stage before writing to the next queue. This still creates an intra-stage synchronization barrier, especially in rollout and evaluate stages where a minibatch contains many LLM requests. To reduce this barrier, FlashEvolve allows a stage to release partial output after a fraction $\alpha^i_{\text{spec}} \in (0, 1]$ of its requests has finished. The worker packages the completed results as a tentative queue item and continues the remaining requests in the background, while downstream workers can start from the tentative item.

For rollout, this means completed samples can be forwarded as soon as they are available. For evaluation, FlashEvolve adds a score threshold to avoid forwarding weak candidates. After the first α^i_{spec} fraction of evaluation requests finishes, the worker computes a partial score. If the partial score exceeds the current pool score, FlashEvolve inserts the candidate into the pool as a *speculative artifact*. When full evaluation finishes, the artifact is confirmed if it still passes the acceptance condition; otherwise, it is removed. If a speculative artifact is later removed, downstream items derived from it are marked stale and handled by the same staleness-aware policy in Section 3.2; they cannot update the confirmed pool without passing the normal validation path.

Validation-set reordering. Speculative completion is more reliable when the early validation samples are informative. We call the first α_{spec} fraction of the validation set the *speculative prefix*. FlashEvolve reorders the validation set using sample pass history: samples that pass for w consecutive rounds are moved out of the speculative prefix and placed later in the validation order. This keeps easy samples from dominating the early signal and leaves more discriminative samples in the prefix. We set $w = 3$ to avoid reacting to one-round noise while keeping the prefix responsive to artifact improvement.

3.4 Adaptive Workflow Control

Different stages in an evolution loop produce and consume items at different rates. A stage with short LLM requests can quickly fill the queue of a later stage whose requests are longer or more imbalanced. If workers keep running at a fixed concurrency, the queue keeps growing and many items become stale before they are processed. FlashEvolve therefore monitors queue pressure and version gap to adjust worker behavior, making execution more balanced and token efficient.

Adaptive worker reallocation. FlashEvolve measures the item production rate of each asynchronous stage. The production rate is the number of queue items that a stage writes to its downstream queue per second. A stage with a much lower production rate can limit the whole workflow, while a stage with a much higher production rate can overfeed downstream queues.

FlashEvolve compares production rates across stages and adjusts their worker counts. If a stage produces items at less than half the median stage rate, we increase its worker count. If a stage produces items at more than twice the median stage rate, we decrease its worker count. Each adjustment

Table 1: Throughput comparison on GEPA workloads. LLM throughput measures the output token rate of the whole system. Proposal throughput measures the rate of new candidate artifact generation.

	LLM Throughput (token/s)				Proposal Throughput (proposal/min)			
Method	IFBench	HotpotQA	HoVer	AIME	IFBench	HotpotQA	HoVer	AIME
vLLM with Qwen3-8B								
GEPA	476	30	461	200	1.9	4.6	2.5	2.2
Combee (K=10)	616	38	810	994	1.2	2.7	2.0	6.2
Combee (K=40)	1,035	44	891	977	0.7	4.5	2.0	1.6
FlashEvolve	**4,128**	**93**	**1,255**	**998**	**8.9**	**8.8**	**5.9**	**11.4**
API with GPT-4o-mini								
GEPA	361	14	142	103	1.7	2.4	1.8	1.3
Combee (K=10)	397	18	348	211	1.0	1.4	0.8	1.0
Combee (K=40)	389	23	214	336	0.8	1.2	0.7	0.6
FlashEvolve	**791**	**32**	**352**	**485**	**10.1**	**8.0**	**9.1**	**6.6**

changes the worker count by at most one, and each stage has a minimum and maximum worker count. This avoids large swings while still correcting persistent throughput imbalance.

3.5 Implementation

FlashEvolve is implemented in Python with lightweight threads and in-process queues. Each stage is executed by a small worker pool, queue items carry the artifact state and pool version, and pool updates are applied under a lock. For a fair comparison, we run all open-source baselines and FlashEvolve on the same LLM serving stack: the native LLM calls in different algorithms are replaced by the same `DSPy` client backed by a local vLLM [12] server with an OpenAI-compatible endpoint. Thus all methods benefit from the same continuous batching and KV-cache reuse, and throughput differences mainly reflect the optimization of the evolution pipeline. The same interface is also used for API-based experiments by changing only the endpoint and model name.

4 Evaluation

4.1 Experimental Setup

Evolving algorithm baselines. We evaluate FlashEvolve on three test-time evolution algorithms that optimize different artifacts. We use GEPA [2], which evolves prompts through execution feedback and reflection, as the main algorithm for end-to-end comparison and ablation studies. We set the rollout minibatch size $mb = 3$, following the default setting. We also reproduce Combee [15] as a scaling-oriented baseline. Combee scales batch-level parallelism to improve throughput. We evaluate its reported fixed-batch variants with parallelization sizes $B = 10$ and 40.

We also evaluate FlashEvolve on ACE [35] and Meta-Harness [14]. ACE evolves agent context playbooks, while Meta-Harness evolves harness code. These algorithms use different artifact types, but all fall into the abstraction optimized by FlashEvolve: a multi-stage evolution loop with LLM-heavy stages, queueable intermediate results, and a shared artifact pool that is updated over time.

Models and deployment. For open-source model experiments, we use Qwen3-8B [31], which is the default model used in GEPA and provides a representative setting for studying test-time evolution behavior. We serve Qwen3-8B with vLLM on a single NVIDIA H100 80GB GPU and an AMD EPYC 9534 CPU. For API-based experiments, we use GPT-4o-mini [9], which shows similar evolution behavior to the open-source setting while representing a common remote-serving deployment.

Datasets. We use the benchmark datasets used in each original evolution algorithm. For GEPA, we evaluate on IFBench [24] for instruction following, HotpotQA [32] for knowledge retrieval, HoVer [11] for multi-hop verification, and AIME for mathematical reasoning. For ACE, we use FiNER [17] and FormulaReasoning, which are finance and numerical-reasoning datasets, respectively. For Meta-Harness, we use a mixture of Symptom2Disease for medical diagnosis classification and AGNews [36] for topic categorization. These datasets cover diverse domains of agent applications.

Table 2: Validation score(%) and normalized evolution rate within 30 minutes on GEPA workloads using Qwen3-8B. Evolution rate measures the score improvement achieved within the time budget, normalized by the improvement achieved by serial GEPA, reflecting the speedup of evolution progress. AIME reports "–" because serial GEPA shows no score improvement within the budget.

	IFBench		HoVer		HotpotQA		AIME	
Method	Score	Norm. Rate	Score	Norm. Rate	Score	Norm. Rate	Score	Norm. Rate
GEPA	87.6	1.00	39.8	1.00	**63.3**	**1.00**	10.0	–
Combee (K=10)	88.5	1.39	41.2	1.09	62.5	0.94	10.0	–
Combee (K=40)	86.5	0.55	40.5	1.05	58.6	0.63	10.0	–
FlashEvolve	**90.6**	**2.27**	**42.0**	**1.15**	61.7	0.88	**15.0**	–

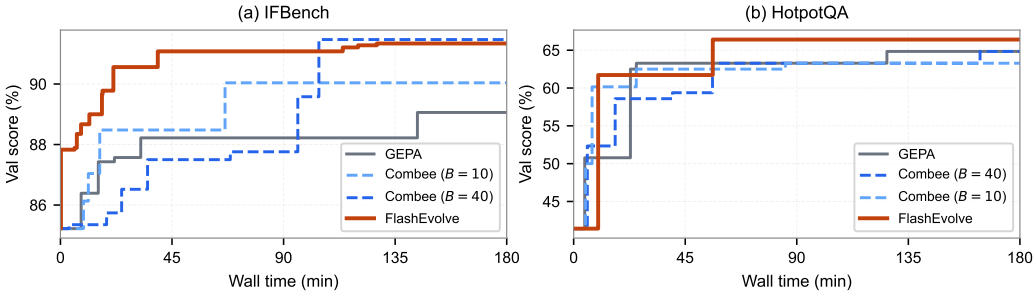


Figure 4: Longer-time validation score evolution over wall-clock time with Qwen3-8B.

4.2 Improvement on GEPA

System throughput improvement. Table 1 first shows that FlashEvolve improves LLM throughput. Within a 30-minute run, asynchronous stage orchestration keeps the LLM backend busier by overlapping requests from different stages and evolution steps. On local vLLM serving, FlashEvolve improves LLM throughput by $4.9\times$ on average over GEPA and $2.1\times$ on average over the best Combee setting. The same trend holds for API-based serving: FlashEvolve improves LLM throughput by $2.9\times$ on average over GEPA and $1.5\times$ on average over the best Combee setting.

We also show that higher LLM throughput translates into faster artifact exploration. On local vLLM serving, FlashEvolve improves proposal throughput by $3.5\times$ on average over GEPA and $3.5\times$ on average over the best Combee setting. On API-based serving, FlashEvolve improves proposal throughput by $4.9\times$ on average over GEPA and $8.4\times$ on average over the best Combee setting. Across all settings, FlashEvolve sustains more than 5.9 proposals/min and up to 11.4 proposals/min, showing that it substantially increases the rate at which evolution tests new candidate artifacts.

Evolving efficiency improvement. Table 2 reports validation score and normalized evolution rate within a fixed 30-minute budget. Across the three workloads where the GEPA baseline makes measurable progress, FlashEvolve achieves an average normalized evolution rate of $1.43\times$. The strongest gain appears on IFBench, where FlashEvolve improves the validation score from 87.6 to 90.6 and reaches a $2.27\times$ normalized evolution rate. On HoVer, FlashEvolve also achieves the best score and a $1.15\times$ normalized rate. On HotpotQA, FlashEvolve does not reach the best validation score within the 30-minute budget, but Figure 4 shows that this advantage emerges under a longer budget. On AIME, both GEPA and Combee remain at the initial score of 10.0%, while FlashEvolve reaches 15.0%, making it the only method that improves over the initial score.

Long-time evolution. Figure 4 reports validation score over a longer 180-minute budget. FlashEvolve reaches strong validation scores earlier than the synchronous baselines. On IFBench, FlashEvolve reaches 91% in 39.3 minutes, while Combee (B=40) reaches the same score region after 104.2 minutes and eventually approaches a similar final score. On HotpotQA, FlashEvolve reaches its best score of 66.41% at 56.1 minutes and maintains the highest validation score over the full budget, while all baselines remain below 65%. These results show that asynchronous evolution accelerates useful artifact discovery, and in some workloads also improves the final score under a longer time budget.

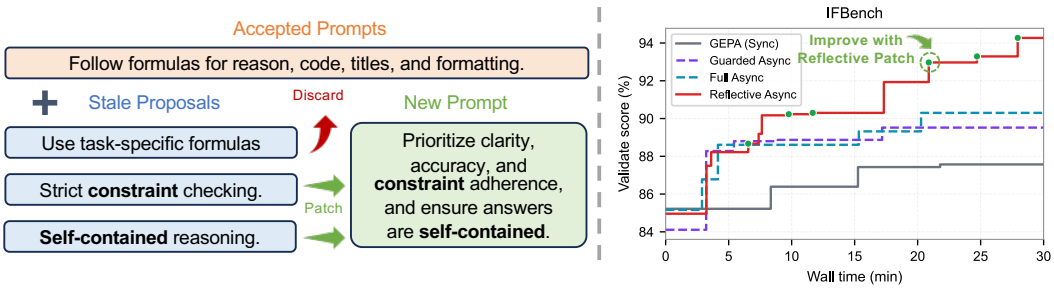


Figure 5: Staleness handling on IFBench with Qwen3-8B. The left panel shows an example of reflective prompt repair. The right panel compares Full Async, Guarded Async, and Reflective Async.

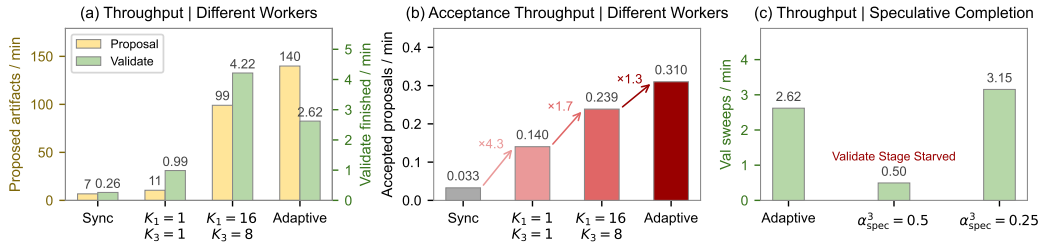


Figure 6: Ablation study on IFBench with Qwen3-8B. (a) Worker allocation varies proposed artifact throughput (before minibatch evaluation). (b) Adaptive worker control achieves the highest accepted proposal throughput by balancing proposal generation and validation. (c) Speculative completion improves validation throughput when the prefix threshold is properly set.

4.3 Ablation Studies

Comparison across staleness handling methods. Figure 5 compares three staleness handling policies on IFBench. Full Async and Guarded Async achieve higher final scores than the synchronous pipeline. Full Async preserves all stale items and therefore keeps high throughput, while Guarded Async discards highly stale items to avoid outdated updates.

Reflective Async achieves the best evolution efficiency. It reaches a validation score of 94.3% within the 30-minute budget. This shows that stale items are not always useless: when the stale artifact is text, FlashEvolve can inspect it, discard task-specific noise, and reuse transferable principles. Figure 5 also shows a simplified repair example extracted from our logs. FlashEvolve discards task-specific formulas because the accepted prompt already contains general instruction-following rules and the formula does not transfer across tasks. In contrast, new takeaways such as stricter constraint checking and self-contained reasoning are distilled into a compact prompt patch. We also observe that many score jumps in the Reflective Async curve come from prompts after such patches, suggesting that reflective repair improves the quality of prompt proposals rather than only increasing throughput.

Concurrent Workers. We study how worker counts affect stage throughput. As shown in Figure 6(a), larger worker counts greatly increase proposal throughput, from 7 artifacts/min in the synchronous setting to 99 artifacts/min with K_1=16, K_3=8. However, validation throughput does not scale uniformly, showing that naive scaling can shift the bottleneck across stages. Adaptive control balances queue pressure and stage rates; although it reduces validation throughput compared with the fixed large-worker setting, it achieves the highest accepted proposal throughput in Figure 6(b), suggesting that a more balanced worker allocation can produce a higher rate of high-quality candidates rather than merely increasing raw proposal volume.

Speculative Stage Completion. Figure 6(c) shows that speculative completion can improve validation throughput when the prefix threshold is properly set. With α^3_{spec}=0.25, validation-stage throughput increases to 3.15 validations/min and the validation score improves by 4.49 percentage points within 30 minutes. However, when α^3_{spec}=0.5, the speculative gate becomes less effective: candidates must wait for a larger partial validation prefix before being released, which reduces early handoff and

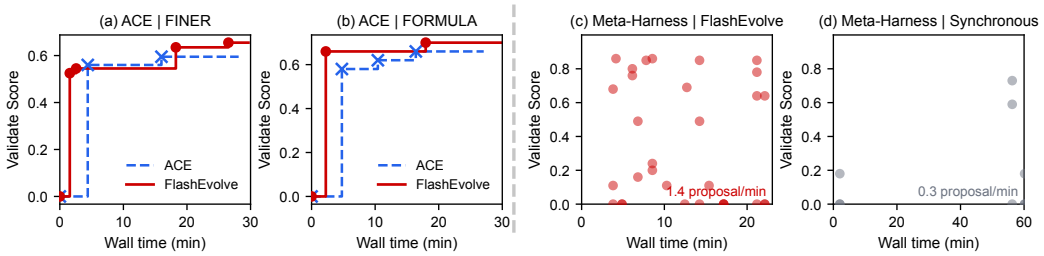


Figure 7: FlashEvolve on other algorithms for agent evolution. (a) and (b) compare validation score evolution curve on FiNER and Formula over a fixed time budget. (c) and (d) compare Meta-Harness proposal rate and validation scores of different proposals on Symptom2Disease and AgNews.

lowers validation throughput. Overall, speculative completion can be useful in some settings, but its accuracy and efficiency depend on α_{spec} and dataset characteristics. We therefore treat it as an optional optimization in FlashEvolve rather than include it in the main evaluation.

4.4 Improvement on Other Algorithms: ACE and Meta-Harness

FlashEvolve is algorithm-agnostic. It does not rely on a specific artifact type, but only assumes that the evolution loop contains multiple stages that need orchestration. We evaluate FlashEvolve on two other algorithms including ACE [35] and Meta-Harness [14], which evolves harness code.

Figure 7(a)(b) shows the wall-clock evolution curves of ACE. FlashEvolve reaches better validation scores within the same 30-minute budget on both tasks. The validation score improves from 0.6 to 0.66 on FiNER and from 0.66 to 0.7 on Formula, demonstrating higher efficiency.

Figure 7(c)(d) compares FlashEvolve with the synchronous Meta-Harness baseline. FlashEvolve improves the proposal and validation throughput from 0.3 to 1.4 proposals/min, giving a $4.7\times$ speedup. Since the open-source model has relatively weak code-generation capability, harness-code evolution progresses slowly in both settings. We therefore report the score distribution of different proposed harnesses. With higher proposal throughput, FlashEvolve samples and validates more harness candidates within the same time budget, leading to a higher potential of improvement.

4.5 Token Usage and API Costs

We evaluate token and monetary costs on IFBench for the local Qwen3-8B model under the same 30-minute wall-clock budget. Table 3 reports the results for Qwen3-8B. FlashEvolve processes $8.8\times$ more total tokens within the same wall clock budget. This is expected because asynchronous execution uses GPU capacity that synchronous GEPA leaves idle to complete additional proposal generation and evaluation work. Reflective repair contributes only 0.16M tokens, and is not the primary source of the increased token consumption.

Table 3: Token consumption for local Qwen3-8B on IFBench within 30-minute budget.

Method	Input tokens	Output tokens	Reflection tokens	Total tokens
Synchronous GEPA	2.01M	0.86M	0	2.87M
FlashEvolve	17.84M	7.43M	0.16M	25.43M

Table 4 compares synchronous GEPA and FlashEvolve on IFBench with GPT-4o-mini under a matched budget of approximately 2,000 API calls. This comparison controls the number of API calls rather than elapsed time. Under the matched call budget, FlashEvolve uses 2.57M tokens and costs $0.71, compared with 2.36M tokens and $0.68 for synchronous GEPA, while improving validation from 80.47 to 83.40. The average token usage per call differs by less than 1 percent.

Table 4: Token and API costs for GPT-4o-mini on IFBench under approximately 2,000 API calls.

Method	Input tokens	Output tokens	Total tokens	Tokens/call	API cost	Validation (%)
Synchronous GEPA	1.63M	0.73M	2.36M	1188	$0.68	80.47
FlashEvolve	1.84M	0.73M	2.57M	1197	$0.71	83.40

4.6 CPU and Memory Usage

Table 5 measures the CPU and memory overhead of runtime orchestration on IFBench with Qwen3-8B as the concurrent worker count increases from $K = 1$ to 4, 16, and 64. It reports average CPU cores, CPU usage, and peak memory, using process CPU time and peak-memory measurements.

Table 5: CPU and memory overhead of runtime orchestration.

K	Average CPU cores	CPU usage	Peak memory
1	0.034	3.4%	1.37 GB
4	0.095	9.5%	1.52 GB
16	0.548	54.8%	1.69 GB
64	0.866	86.6%	1.98 GB

Even at $K = 64$, the runtime uses less than one CPU core and approximately 2 GB of memory while the GPU remains high utilization. Most of the memory is the fixed PyTorch and DSPy environment. The additional memory comes mainly from worker threads and active artifacts. These results show that runtime orchestration is not the performance or memory bottleneck.

5 Conclusion

This paper presents FlashEvolve, a framework for accelerating agent evolution in wall-clock time. FlashEvolve replaces synchronous stage execution with asynchronous workers and queues, allowing LLM-heavy stages and evolution steps to overlap. It preserves evolution semantics through artifact-pool versioning and staleness-aware policies that update, discard, or patch stale artifacts, and further improves efficiency with speculative stage completion and adaptive workflow control. On GEPA workloads, FlashEvolve achieves $3.5\times$ higher proposal throughput over the synchronous implementation on local vLLM serving. The same execution model also generalizes to context evolution with ACE and harness-code evolution with Meta-Harness.

Limitations. FlashEvolve currently supports only a limited set of agent-evolution algorithms, and each integration still requires algorithm-specific implementation effort. Although the worker and queue abstraction is general, mapping a new algorithm to this abstraction requires implementing its stages, queue items, artifact state, and update rules. Our current evaluation also focuses on representative prompt, context, and harness-code evolution workloads, and broader coverage of evolution algorithms and artifact types remains future work.

Future Work. In future work, we plan to expand FlashEvolve with a more general plugin interface for defining stages, artifacts, staleness policies, and evaluation logic, so that new evolution algorithms can be integrated with less manual engineering. We also plan to extend FlashEvolve to more types of artifact evolution, such as memory, tool-use policies and generated programs.

References

[1] Nemo rl: A scalable and efficient post-training library. `https://github.com/NVIDIA-NeMo/RL`, 2025. GitHub repository.

[2] L. A. Agrawal, S. Tan, D. Soylu, N. Ziems, R. Khare, K. Opsahl-Ong, A. Singhvi, H. Shandilya, M. J. Ryan, M. Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

[3] H. Assumpção, D. Ferreira, L. Campos, and F. Murai. Codeevolve: An open source evolutionary coding agent for algorithm discovery and optimization. *arXiv preprint arXiv:2510.14150*, 2025.

[4] J. Fang, Y. Peng, X. Zhang, Y. Wang, X. Yi, G. Zhang, Y. Xu, B. Wu, S. Liu, Z. Li, et al. A comprehensive survey of self-evolving ai agents: A new paradigm bridging foundation models and lifelong agentic systems. *arXiv preprint arXiv:2508.07407*, 2025.

[5] W. Fu, J. Gao, X. Shen, C. Zhu, Z. Mei, C. He, S. Xu, G. Wei, J. Mei, J. Wang, et al. Areal: A large-scale asynchronous reinforcement learning system for language reasoning. *arXiv preprint arXiv:2505.24298*, 2025.

[6] H.-a. Gao, J. Geng, W. Hua, M. Hu, X. Juan, H. Liu, S. Liu, J. Qiu, X. Qi, Y. Wu, et al. A survey of self-evolving agents: On path to artificial super intelligence. *arXiv preprint arXiv:2507.21046*, 1, 2025.

[7] D. Guo, D. Yang, H. Zhang, J. Song, P. Wang, Q. Zhu, R. Xu, R. Zhang, S. Ma, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[8] Z. Hu, H. Ouyang, C. Chen, Z. Pan, Y. Guan, Z. Yu, Z. Wang, S. Swanson, and Y. Ding. Jigsawrl: Assembling rl pipelines for efficient llm post-training, 2026. URL `https://arxiv.org/abs/2604.23838`.

[9] A. Hurst, A. Lerer, A. P. Goucher, A. Perelman, A. Ramesh, A. Clark, A. Ostrow, A. Welihinda, A. Hayes, A. Radford, et al. Gpt-4o system card. *arXiv preprint arXiv:2410.21276*, 2024.

[10] A. Jaech, A. Kalai, A. Lerer, A. Richardson, A. El-Kishky, A. Low, A. Helyar, A. Madry, A. Beutel, A. Carney, et al. Openai o1 system card. *arXiv preprint arXiv:2412.16720*, 2024.

[11] Y. Jiang, S. Bordia, Z. Zhong, C. Dognin, M. Singh, and M. Bansal. Hover: A dataset for many-hop fact extraction and claim verification. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 3441–3460, 2020.

[12] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[13] R. T. Lange, Y. Imajuku, and E. Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

[14] Y. Lee, R. Nair, Q. Zhang, K. Lee, O. Khattab, and C. Finn. Meta-harness: End-to-end optimization of model harnesses. *arXiv preprint arXiv:2603.28052*, 2026.

[15] H. Li, R. He, Q. Zhang, C. Ji, Q. Mang, X. Chen, L. A. Agrawal, W.-L. Liao, E. Yang, A. Cheung, et al. Combee: Scaling prompt learning for self-improving language model agents. *arXiv preprint arXiv:2604.04247*, 2026.

[16] X. Lou, M. Lázaro-Gredilla, A. Dedieu, C. Wendelken, W. Lehrach, and K. P. Murphy. Auto-harness: improving llm agents by automatically synthesizing a code harness. *arXiv preprint arXiv:2603.03329*, 2026.

[17] L. Loukas, M. Fergadiotis, I. Chalkidis, E. Spyropoulou, P. Malakasiotis, I. Androutsopoulos, and G. Paliouras. Finer: Financial numeric entity recognition for xbrl tagging. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 4419–4431, 2022.

[18] H. Lu, H. Huang, Y. Zhou, C. Li, and N. Zhu. Empirical-mcts: Continuous agent evolution via dual-experience monte carlo tree search. *arXiv preprint arXiv:2602.04248*, 2026.

[19] A. Madaan, N. Tandon, P. Gupta, S. Hallinan, L. Gao, S. Wiegreffe, U. Alon, N. Dziri, S. Prabhumoye, Y. Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in neural information processing systems*, 36:46534–46594, 2023.

[20] A. Novikov, N. Vũ, M. Eisenberger, E. Dupont, P.-S. Huang, A. Z. Wagner, S. Shirobokov, B. Kozlovskii, F. J. Ruiz, A. Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

[21] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, et al. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[22] S. Ouyang, J. Yan, I. Hsu, Y. Chen, K. Jiang, Z. Wang, R. Han, L. T. Le, S. Daruki, X. Tang, et al. Reasoningbank: Scaling agent self-evolving with reasoning memory. *arXiv preprint arXiv:2509.25140*, 2025.

[23] Z. Pan, A. D. PATEL, Y. Shen, Z. Hu, Y. Guan, W.-L. Li, L. Qin, Y. Wang, and Y. Ding. Kvflow: Efficient prefix caching for accelerating llm-based multi-agent workflows. *Advances in Neural Information Processing Systems*, 38:126246–126265, 2026.

[24] V. Pyatkin, S. Malik, V. Graf, H. Ivison, S. Huang, P. Dasigi, N. Lambert, and H. Hajishirzi. Generalizing verifiable instruction following. *arXiv preprint arXiv:2507.02833*, 2025.

[25] G. Sheng, Y. Tong, B. Wan, W. Zhang, C. Jia, X. Wu, Y. Wu, X. Li, C. Zhang, Y. Peng, et al. Laminar: A scalable asynchronous rl post-training framework. *arXiv preprint arXiv:2510.12633*, 2025.

[26] G. Sheng, C. Zhang, Z. Ye, X. Wu, W. Zhang, R. Zhang, Y. Peng, H. Lin, and C. Wu. Hybridflow: A flexible and efficient rlhf framework. In *Proceedings of the Twentieth European Conference on Computer Systems*, pages 1279–1297, 2025.

[27] N. Shinn, F. Cassano, A. Gopinath, K. Narasimhan, and S. Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36: 8634–8652, 2023.

[28] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, and B. Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

[29] X. Wang, C. Li, Z. Wang, F. Bai, H. Luo, J. Zhang, N. Jojic, E. P. Xing, and Z. Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *arXiv preprint arXiv:2310.16427*, 2023.

[30] E. Xiao, Y. Zeng, A. Chen, C.-J. Li, A. Bertsch, and G. Neubig. Prompt-mii: Meta-learning instruction induction for llms. *arXiv preprint arXiv:2510.16932*, 2025.

[31] A. Yang, A. Li, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Gao, C. Huang, C. Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[32] Z. Yang, P. Qi, S. Zhang, Y. Bengio, W. Cohen, R. Salakhutdinov, and C. D. Manning. Hotpotqa: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 conference on empirical methods in natural language processing*, pages 2369–2380, 2018.

[33] M. Yuksekgonul, F. Bianchi, J. Boen, S. Liu, Z. Huang, C. Guestrin, and J. Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

[34] G. Zhang, H. Ren, C. Zhan, Z. Zhou, J. Wang, H. Zhu, W. Zhou, and S. Yan. Memevolve: Meta-evolution of agent memory systems. *arXiv preprint arXiv:2512.18746*, 2025.

[35] Q. Zhang, C. Hu, S. Upasani, B. Ma, F. Hong, V. Kamanuru, J. Rainton, C. Wu, M. Ji, H. Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

[36] X. Zhang, J. Zhao, and Y. LeCun. Character-level convolutional networks for text classification. *Advances in neural information processing systems*, 28, 2015.

[37] Y. Zhao, A. Gu, R. Varma, L. Luo, C.-C. Huang, M. Xu, L. Wright, H. Shojanazeri, M. Ott, S. Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

[38] L. Zheng, L. Yin, Z. Xie, C. Sun, J. Huang, C. H. Yu, S. Cao, C. Kozyrakis, I. Stoica, J. E. Gonzalez, et al. Sglang: Efficient execution of structured language model programs. *Advances in neural information processing systems*, 37:62557–62583, 2024.

[39] Y. Zhong, Z. Zhang, X. Song, H. Hu, C. Jin, B. Wu, N. Chen, Y. Chen, Y. Zhou, C. Wan, et al. Streamrl: Scalable, heterogeneous, and elastic rl for llms with disaggregated stream generation. *arXiv preprint arXiv:2504.15930*, 2025.

A Evolution Quality under Different Staleness Policies

A.1 Validation Scores versus Rollouts

Table 6 compares the validation scores of synchronous GEPA, Full Async, Guarded Async, and Reflective Async on IFBench with Qwen3-8B at matched cumulative rollout budgets from 2,000 to 10,000. This single-run comparison examines validation progress per rollout, complementing the wall-clock evaluation in the main text. Up to approximately 4000 rollouts, all four modes remain within about one validation point. This shows that asynchronous execution does not materially reduce validation progress at a matched rollout budget. At larger budgets, synchronous GEPA, Full Async, and Guarded Async plateau, while Reflective Async continues to improve.

Table 6: Single-run IFBench comparison for staleness-aware policies at matched rollout budgets.

Rollouts	2,000	4,000	6,000	8,000	10,000
Synchronous GEPA	89.1	90.1	90.1	90.1	90.1
Full Async	88.6	90.3	90.3	90.3	90.3
Guarded Async	88.8	89.5	89.5	89.5	89.5
Reflective Async	88.7	90.3	91.9	93.3	94.3

A.2 Reflection in Synchronous and Asynchronous GEPA

Table 7 separates the effects of reflective synthesis and asynchronous execution by comparing synchronous GEPA, Full Async, synchronous GEPA with reflection, and Reflective Async on IFBench with Qwen3-8B. Each run takes 40 to 60 minutes to exhaust the rollout budget. It reports validation scores at matched rollout budgets and proposal throughput over six seeds. For the synchronous reflection control, we implemented the same reflective synthesis procedure without asynchronous execution. At a fixed interval, recent proposals are collected as reflection material. The same repair prompt then synthesizes their useful information into the current frontier, and the resulting candidate enters the standard validation stage. This control tests whether reflection provides value even without asynchronous execution.

Table 7: IFBench validation score (%) at matched rollout budgets and proposal throughput with Qwen3-8B.

Method	2,000	4,000	6,000	8,000	10,000	Proposals/min
Synchronous GEPA	90.0 ± 1.1	90.1 ± 0.8	90.3 ± 0.6	90.6 ± 0.8	90.8 ± 0.9	3.9 ± 0.3
Full Async	88.6 ± 0.6	89.7 ± 0.5	89.9 ± 0.7	90.7 ± 0.8	90.9 ± 0.3	19.6 ± 3.2
Synchronous + reflection	89.3 ± 0.8	90.0 ± 0.9	90.6 ± 0.3	90.9 ± 0.5	91.3 ± 0.5	3.6 ± 0.1
Reflective Async	89.0 ± 0.7	89.5 ± 0.6	91.1 ± 0.7	91.4 ± 1.1	91.4 ± 1.1	22.6 ± 1.9

Reflection provides a modest positive trend in validation quality. This improvement also appears when reflection is added to synchronous GEPA. The likely reason is that the additional LLM call extracts shared information from several proposals and incorporates it into the current frontier. Without reflection, this information would remain unused or be discarded.

A.3 Quality of Stale and Nonstale Proposals

Table 8 compares the minibatch pass rates and full-validation acceptance rates of stale and nonstale GEPA proposals on IFBench, HoVer, and HotpotQA with Qwen3-8B. This experiment examines where staleness reduces the fraction of proposals that produce useful frontier updates. A proposal generated from an outdated prompt is often no longer useful for the current frontier. Such proposals are less likely to pass the minibatch evaluation and tend to perform worse during full validation. The computation used to generate and evaluate them is therefore wasted.

Stale proposals consistently have lower minibatch pass rates and much lower full validation acceptance rates. The averaged full validation acceptance rate shows significant drop, showing that most stale proposals do not produce useful frontier updates.

Table 8: Quality of stale and nonstale GEPA proposals with Qwen3-8B.

Dataset	Proposal	Minibatch pass rate	Full validation acceptance rate
IFBench	no staleness	32.4%	22.2%
IFBench	stale proposals	31.7%	10.3%
HoVer	no staleness	24.3%	28.6%
HoVer	stale proposals	20.3%	2.4%
HotpotQA	no staleness	26.7%	10.0%
HotpotQA	stale proposals	23.1%	0.0%

A.4 GEPA Validation Scores and Wall-clock Times after Five Epochs

Table 9 compares the validation scores and wall-clock times of synchronous GEPA, Full Async, and Reflective Async after five evolution epochs on HotpotQA and IFBench with Qwen3-8B. Unlike the fixed-time and fixed-rollout comparisons above, this experiment evaluates evolution quality after the same number of passes over the training data.

Table 9: GEPA validation score (%) and elapsed time after five evolution epochs with Qwen3-8B.

Task	Method	Validation score	Time (min)
HotpotQA	Synchronous GEPA	62.50	77
HotpotQA	Full Async	61.72	24
HotpotQA	Reflective Async	64.84	35
IFBench	Synchronous GEPA	88.74	70
IFBench	Full Async	88.35	11
IFBench	Reflective Async	89.26	17

Full Async performs worse than Synchronous GEPA after five epochs, suggesting that unconstrained asynchronous execution introduces a loss in evolution quality. This means that more epochs are required to achieve the same validation score. Given that Synchronous GEPA achieves meaningful gains on both IFBench and HotpotQA, the corresponding loss from Full Async is not negligible. Reflective Async closes this gap through reflective repair.

B Staleness Handling Overhead

To assess staleness handling policy overhead, we evaluate Guarded Async and Reflective Async over five seeds on IFBench with Qwen3-8B. We report the proposal discard rate, the fraction of total tokens spent on discarded proposals, and the fraction spent on reflective repair as mean $\pm$ standard deviation and observed ranges.

For Guarded Async, the discard rate is run dependent at $11.5\% \pm 15.7\%$, ranging from 0% to 40%, because it depends on how frequently the frontier advances. However, discarded proposals account for only $0.34\% \pm 0.47\%$ of total tokens, with a range of 0% to 1.2%. Even in the run with a 40% discard rate, only 1.2% of the token budget was wasted. Thus, the discard frequency varies across runs, but its token overhead remains consistently small.

For Reflective Async, repair calls consume $0.49\% \pm 0.35\%$ of total tokens, ranging from 0.10% to 0.91%. They run in a background worker and introduce no synchronization barrier. Reflective Async therefore reuses information from proposals that Guarded Async would discard, while adding less than 1% token overhead in every seed. The resulting quality benefit depends on the dataset and the remaining room for frontier improvement.

C Speculative Completion Costs

To quantify rollback frequency and token overhead from speculative completion, we evaluate IFBench with Qwen3-8B using a validation set of 32 questions. We measure how often an early-completed candidate is rejected by full validation and the fraction of total tokens associated with the resulting

interim work. FlashEvolve proposes a candidate after partial validation so that subsequent work can begin before all 32 questions finish.

About 1/5 early completed candidates was later rejected by the complete validation result, corresponding to approximately one rollback per 5 speculative validation stages. The associated tokens account for only 0.6 to 0.8 percent of the total because an incorrect early release affects proposal generation only during the short interval before the full validation. Most tokens are still consumed by the full validation stage, which continues normally regardless of the early result.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and introduction state that FlashEvolve improves wall-clock efficiency for agent evolution through asynchronous stage orchestration, staleness-aware handling, speculative completion, and adaptive workflow control. These claims are supported by the experimental results on GEPA, Combee, ACE, and Meta-Harness.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The paper includes a limitations discussion after the conclusion. It discusses the current scope of supported evolution algorithms and the dependence on customized implementations for each target codebase.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [N/A]

Justification: The paper does not present formal theoretical results, theorems, or proofs. Its contributions are systems mechanisms and empirical evaluation.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: The paper describes the evaluated algorithms, datasets, models, serving backends, hardware platform, and key configuration choices used in the main experiments. These details provide the information needed to reproduce the experiment.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
  (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
  (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
  (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
  (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in

some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: We provide an anonymized code repo with instructions for setup. Code is available at `https://anonymous.4open.science/r/FlashEvolve-FEC7/`.

   Guidelines:

   - The answer [N/A] means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: The paper specifies the evaluated evolution algorithms, datasets, models, serving backends, minibatch settings, worker settings, hardware platform, and evaluation metrics used in the main experiments. Additional details are provided in the code repository.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: The reported system-efficiency results are obtained from repeated measurements under the same experimental settings, and the tables report mean measured values for each evaluated workload.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.

- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: The paper reports the main compute resources used in the experiments, including the local setup and API-based experiments. It also reports wall-clock time or fixed time budgets for the main workloads.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The research follows the NeurIPS Code of Ethics. It uses public benchmarks and existing LLM backends for systems evaluation, and we preserve anonymity in the submission.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

Answer: [Yes]

Justification: The paper discusses that FlashEvolve can reduce the cost of agent evolution and provides a system that is easier to use and deploy.

Guidelines:

- The answer [N/A] means that there is no societal impact of the work performed.
- If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

Answer: [N/A]

Justification: The paper does not release new pretrained models, image generators, or scraped datasets with high misuse risk. It evaluates systems mechanisms using existing models and public benchmark datasets.

Guidelines:

- The answer [N/A] means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: The paper cites the existing algorithms, models, serving systems, and benchmark datasets used in the evaluation. We follow their intended research use and provide license and access information in the anonymized artifact.

Guidelines:

- The answer [N/A] means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: The paper releases FlashEvolve as a new code artifact. The anonymized artifact includes documentation, configuration files, and instructions for reproducing the main experiments.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing experiments or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

Answer: [N/A]

Justification: The paper does not involve crowdsourcing experiments or research with human subjects, so IRB approval or equivalent review is not applicable.

Guidelines:

- The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

Answer: [Yes]

Justification: LLMs are central to the evaluated agent-evolution workflows. The paper describes the LLM backends used in the experiments, including Qwen3-8B with vLLM and GPT-4o-mini through API serving.

Guidelines:

- The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
- Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.