PACE: Two-Timescale Self-Evolution for Small Language Model Agents

Chen Ling*, **Pei Chen, Xiangchen Guan, Jiaming Qu**
Shayan Ali Akbar, Madhu Gopinathan, Erwin Cornejo
Amazon

Abstract

Deploying language-model agents in production often requires substantial compute and human effort to tune prompts, parsers, validators, and other components of the agent pipeline. Self-evolution offers a promising alternative, but most existing frameworks assume access to frontier models that can reliably diagnose failures, propose revisions, and judge their own updates. We study whether frozen small language models (SLMs) can serve as effective self-evolving agents under resource constraints. We propose PACE (Prompt And Control Logic Evolution), a two-timescale framework that coordinates low-risk prompt refinement with higher-risk control-logic updates. PACE evolves prompts under fixed control logic until prompt-level gains saturate, then considers constrained control-logic updates that are accepted through held-out validation. Across three frozen SLM backbones ranging from 4B to 14B parameters and four controlled benchmarks, PACE achieves the best performance on all 12 backbone–benchmark combinations. On MMLU and IFEval, it improves over vanilla SLM agents by up to $+9.2\%$ relative improvement and over the stronger single-mode evolution baseline by up to 5.4 percentage points. A τ-bench case study further shows that PACE improves multi-turn tool-use success over vanilla and prompt-only evolution. These results suggest that reliable SLM agent self-evolution is possible without updating model weights or relying on frontier-model teachers, and that the key benefit is not any single final solver pattern but autonomous, validated discovery of task-appropriate inference strategies. Code is available at `https://github.com/lingchen0331/PACE-self-evolving-agent`.

1 Introduction

Language-model-based agents [Wang et al., 2024a] have become a common abstraction for solving complex tasks through reasoning, tool use, verification, and iterative refinement. However, deploying such agents in production often requires substantial compute and repeated human intervention to tune prompts, parsers, validators, and other components of the agent pipeline. Recent work on agent self-evolution [Agrawal et al., 2025, Opsahl-Ong et al., 2024, Zhang et al., 2025] offers a promising alternative: agents can improve their own behavior by using execution feedback to revise *prompts/task contexts*, or modify *control logic* without changing the underlying model parameters. Most existing approaches, however, assume access to strong frontier models that can reliably diagnose failures, propose high-quality revisions, and judge whether those revisions should be accepted. These assumptions become fragile when the agent is powered by a small language model (SLM).

This paper studies self-evolution for frozen SLM agents. We focus on models with at most 14B parameters, where the model weights remain fixed throughout the evolution process. This setting is important for practical deployments in which local serving, latency, privacy, or cost constraints make

*Corresponding author: `lingchen9531@gmail.com`. Work conducted at Amazon; current affiliation: Reddit.

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

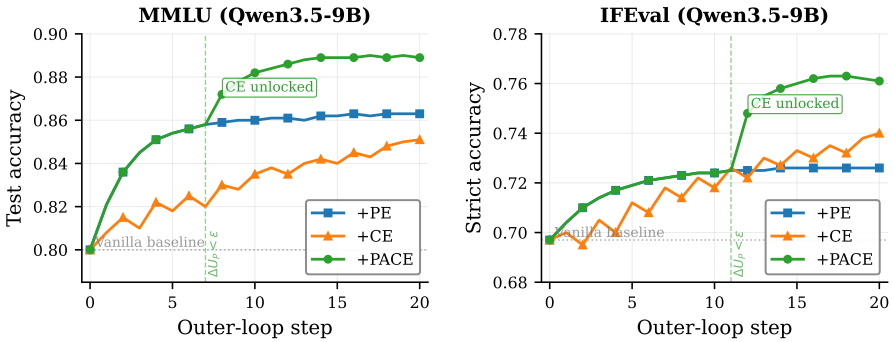

Figure 1: PACE evolution dynamics on Qwen3.5-9B. Prompt-only evolution (+PE) improves early but quickly saturates, while control-logic evolution (+CE) is noisy when structural updates are applied from the start. PACE first exploits stable prompt refinement, then introduces a validated control-logic update after prompt gains plateau, producing a performance jump and final better accuracy.

frontier-model APIs or large-scale fine-tuning undesirable. It is also technically challenging. SLMs are more sensitive to prompt complexity and often reach diminishing returns after a small number of prompt revisions. At the same time, allowing an SLM to freely rewrite its own executable control logic[2] can be unstable: proposed edits may be syntactically valid but semantically incorrect, causing silent regressions in parsing, validation, retry behavior, or inference-time decision rules.

A key observation motivating our work is that prompt updates and control-logic updates play different roles in agent improvement. As shown in Figure 1, prompt-only evolution improves performance quickly but often saturates once the remaining failures stem from structural bottlenecks, such as brittle output extraction, missing validation, weak repair logic, or ineffective sampling policies. In contrast, enabling control-logic updates from the beginning is substantially less stable, since SLM-proposed structural edits can introduce regressions before the prompt has been sufficiently optimized. These results suggest that reliable SLM self-evolution should not treat prompt refinement and control-logic modification as interchangeable actions: it should first exploit low-risk prompt updates, then invoke higher-risk control-logic updates only after prompt-level gains plateau, with an explicit mechanism for validating proposed changes.

We propose **PACE** (**P**rompt **A**nd **C**ontrol Logic **E**volution), a two-timescale agentic framework for self-evolving frozen SLM. PACE operationalizes self-evolution through a controller that can invoke multiple adaptation tools, including prompt evolution, failure analysis, control logic proposal, control logic validation, etc. Prompt evolution is treated as a frequently callable, low-risk tool under a fixed agent structure, while control-logic evolution is treated as a higher-risk adaptation action that is considered if prompt-level gains saturate. To make such updates reliable, PACE separates *proposal* from *acceptance*: the SLM may propose constrained changes to safe-to-edit components of the agent pipeline, but a candidate structure is committed only if it improves over the current agent in a held-out validation while satisfying the resource budget. PACE is not merely prompt optimization plus code editing; the contribution is instead to show that a frozen SLM can autonomously discover when such strategies are useful, propose task-appropriate solver modifications, and commit them only through empirical validation, without human specification of the strategy–task mapping. Our contributions are:

- A two-timescale framework in which measured prompt saturation triggers bounded control-logic search, and accepted updates restart prompt evolution under the new solver.
- A proposal–acceptance separation that uses empirical validation to filter SLM-generated edits without a stronger teacher or weight updates.
- Evaluation across three frozen SLMs and four controlled benchmarks, with ablations, rejected-edit analysis, serving costs, expanded τ-bench baselines, and a larger-model check that delineate when adaptation helps.

[2]Non-parameter code or configuration that governs an agent's inference procedure, including output parsing, validation, retry/repair policies, routing or fallback rules, tool-call handling, and decoding settings. It *excludes* updates to model weights.

2 Related Work

Agent self-evolution [Tao et al., 2024, Fang et al., 2025] has emerged as a vital direction for improving the capabilities of language-model agents. Existing approaches can be broadly categorized according to the portion of the agent definition they are allowed to modify: 1) *prompt-space evolution methods*, which operate purely in the textual domain, and 2) *self-referential control logic evolution methods*, which permit modifications to executable control logic.

Prompt-Space Agent Evolution. Prompt-space evolution improves agent behavior by refining textual artifacts such as system prompts, task instructions, tool descriptions, and output constraints while keeping control logic fixed. These methods use execution feedback, including incorrect outputs, reasoning traces, and tool-use errors, to guide prompt updates [Shinn et al., 2023, Wang et al., 2023]. Since prompt updates remain in natural language, they are stable, sample-efficient, and easy to deploy, especially for frozen or small language models. Recent work [Zhang et al., 2025] further treats prompt evolution as structured search, using reflection, specialization, and Pareto-aware selection to balance performance and cost [Agrawal et al., 2025]. However, prompt-space methods cannot directly repair structural bottlenecks such as brittle parsing, missing validation, or weak retry logic, and therefore often saturate once such failures dominate.

Control Logic Agent Evolution. In contrast, control logic evolution allows agents to inspect and modify their own executable logic [Wang et al., 2024b], including control flow, validation routines, and inference-time configurations. These approaches are often motivated by recursive self-improvement, in which both the agent's policy and its update mechanism evolve jointly through runtime introspection and code modification [Schmidhuber, 2003, Yin et al., 2025, Zhou et al., 2025].

PACE coordinates these established adaptation modes under frozen SLM constraints: prompt refinement is the default action, structural search waits for measured prompt saturation, and a validation gate filters higher-risk executable edits. The contribution is this scheduling and acceptance policy, rather than a new mutation operator or repair primitive. PE-only, edit-space-matched CE-only, transition-threshold, and validation-gate comparisons test its necessity.

3 PACE: Two-Timescale Self-Evolution for Frozen SLM Agents

We consider self-evolution for agents powered by frozen, resource-constrained language models. We first define the agent optimization objective, then introduce PACE as a two-timescale framework for coordinating prompt refinement and constrained control-logic updates.

3.1 Problem Definition and Objective

Let $\mathcal{T}$ denote a distribution over tasks, and let M_θ be a pretrained language model with fixed parameters θ. Throughout evolution, θ remains frozen: the model is not fine-tuned, distilled, or otherwise updated. Adaptation is restricted to the agent definition around the model.

We define an agent as $A = (P, C)$, where P denotes textual artifacts such as system prompts, task instructions, and formatting constraints, and C denotes executable control logic such as parsing routines, validation modules, fallback strategies, and inference-time configurations. Given a task $\tau \sim \mathcal{T}$, the agent produces $y = A(\tau; M_\theta, P, C)$ and is evaluated by a task-specific utility $U(\tau, y)$. Executing the agent incurs a cost $\mathrm{Cost}(A)$, such as latency, token usage, model calls, or API calls. The goal of self-evolution is to iteratively improve (P, C) under a fixed resource budget B:

$$\max_{P,C} \ \mathbb{E}_{\tau\sim\mathcal{T}}\left[U(\tau, A(\tau; M_\theta, P, C))\right] \quad \text{s.t. } \mathrm{Cost}(A) \le B. \tag{1}$$

Because M_θ remains fixed, all performance gains must arise from changes to (P, C).

3.2 PACE: A Two-Timescale Agentic Adaptation Framework

Directly optimizing (P, C) in Eq. (1) is difficult under frozen SLM constraints because prompt edits and control-logic edits fail in different ways. Prompt evolution is inexpensive and stable, but it can saturate once failures stem from structural bottlenecks such as brittle parsing, missing validation, weak retry logic, or ineffective sampling. Control-logic evolution can address these failures by

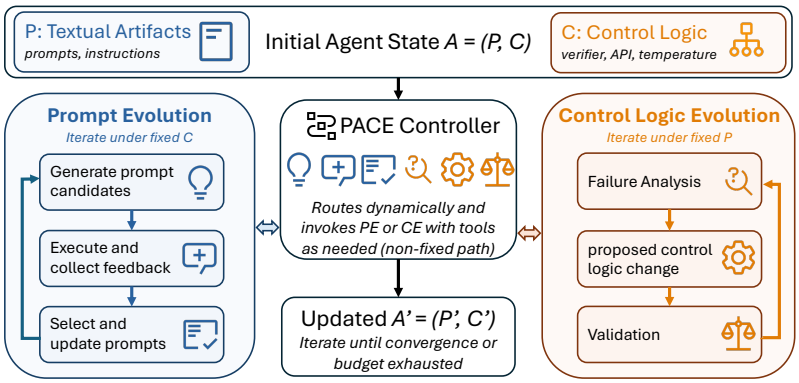

Figure 2: Overview of PACE. An agentic controller invokes prompt evolution until gains saturate, then proposes bounded control-logic updates and commits them only after held-out validation.

changing how model outputs are sampled, parsed, checked, repaired, or re-executed, but such edits are higher-risk: SLM-proposed changes may be non-executable or semantically incorrect.

PACE therefore treats self-evolution as a two-timescale agentic adaptation process rather than an exact joint optimizer over (P, C). As summarized in Figure 2, an agentic controller invokes prompt evolution as the default low-risk tool, uses failure analysis once prompt gains saturate, proposes bounded control-logic updates, and commits a candidate solver only if it improves held-out validation performance under the resource budget. After each accepted update, PACE re-invokes prompt evolution under the new control logic. This cycle coordinates low-risk prompt refinement with higher-risk structural adaptation while reducing premature or harmful control-logic changes.

Prompt Evolution (PE). For a fixed control logic C, PACE uses prompt evolution as the primary low-risk mechanism for improving the current agent. Conceptually, this step searches for a prompt configuration that improves expected task utility under the resource budget:

$$P^*(C) = \arg\max_{P} \ \mathbb{E}_{\tau\sim\mathcal{T}}\left[U(\tau, A(\tau; M_\theta, P, C))\right] \quad \text{s.t. } \mathrm{Cost}(A) \le B. \tag{2}$$

This prompt-evolution tool operates entirely in textual space [Pryzant et al., 2023, Opsahl-Ong et al., 2024]. At each iteration, prompt candidates are produced through three complementary channels: *(i)* a library of handcrafted mutations that perturb the role description, reasoning directives, or sampling temperature; *(ii) reflective* candidates, where the SLM first diagnoses each failure with a one-sentence root-cause analysis (e.g., factual confusion, logical gap, or format error) and a separate proposer call uses these diagnoses to generate targeted prompt revisions; and *(iii) crossover* candidates, in which the two most complementary configurations on the current Pareto front are combined by asking the SLM to unify their respective strengths into a single prompt.

Candidates are evaluated on a small training mini-batch (sampled) and maintained on a Pareto front trading off accuracy against token cost. Rather than always mutating the single highest-accuracy parent, the next parent is sampled from the front with probability proportional to its failure coverage, encouraging exploration of diverse error modes. After all iterations, every front member is re-evaluated on a held-out validation split; the configuration with the best validation accuracy is retained.

Because prompt edits are low-cost, reversible, and stable under limited model capacity, PACE invokes this tool frequently under fixed control logic. Prompt evolution therefore serves as the default adaptation mode before the controller considers higher-risk structural changes.

Constrained Control Logic Evolution (CE). When prompt-level gains saturate, PACE invokes control logic evolution as a higher-risk adaptation tool for modifying the agent's C. Conceptually, this step searches over a restricted structural space $\mathcal{C}$: $\max_{C\in\mathcal{C}} \mathbb{E}_{\tau\sim\mathcal{T}}\left[U(\tau, A(\tau; M_\theta, P^*(C), C))\right]$, where $\mathcal{C}$ denotes the set of admissible control-logic modifications. In practice, control-logic updates are implemented as bounded edits to the agent's solver function, which specifies how the frozen SLM is prompted, sampled, parsed, verified, or re-executed at inference time. These edits are constrained by a fixed solver interface and resource budget, and typically affect safe-to-edit components such as output parsing and validation, retry mechanisms, inference-time configuration, self-consistency or verification passes, and lightweight routing or fallback strategies.

More specifically, PACE guides solver edits using a compact failure summary over recent errors, including extraction/runtime failures, format or constraint violations, and reasoning/content mistakes. The summary steers the next proposal toward an appropriate control-logic change, such as answer-extraction hardening, validation or retry logic, sampling adjustment, self-consistency, or verification/repair. Each candidate solver edit is treated as a proposal and is committed only if it improves held-out performance under the resource budget B. After an accepted update, PACE re-invokes prompt evolution to adapt P to the new solver.

3.3 Empirical Findings for Reliable SLM Agent Evolution

The trajectories (Figure 1), transition and gate ablations (Table 4), failure taxonomy (Figure 3), and proposal filtering (Table 5) motivate two design principles for frozen SLMs. First, the agentic controller must correctly assign improvement opportunities between low-risk prompt refinement and higher-risk solver modification. Second, because SLM-generated control-logic edits can introduce subtle but severe regressions, candidate solver updates should be validated empirically before being committed. We discuss both findings below.

3.3.1 Evolution Credit Assignment

Motivating hypothesis. Under frozen SLMs, prompt evolution often saturates before all error modes are resolved. Further prompt refinement under a fixed control logic C may yield only marginal gains, while admissible solver modifications can still provide non-trivial improvement. The challenge is therefore deciding *which adaptation tool* to invoke when performance stalls: premature control-logic edits are unstable, but excessive prompt updates waste budget once prompt-level gains have saturated.

To describe this tradeoff, we define two conceptual utility gains:

$$\Delta U_P(C) = \max_P \ \mathbb{E}_{\tau\sim\mathcal{T}}\left[U(\tau, A(\tau; M_\theta, P, C))\right] - \mathbb{E}_{\tau\sim\mathcal{T}}\left[U(\tau, A(\tau; M_\theta, P_0, C))\right],$$
$$\Delta U_C(P) = \max_{C'\in\mathcal{C}} \ \mathbb{E}_{\tau\sim\mathcal{T}}\left[U(\tau, A(\tau; M_\theta, P, C'))\right] - \mathbb{E}_{\tau\sim\mathcal{T}}\left[U(\tau, A(\tau; M_\theta, P, C))\right],$$

where P_0 is the baseline prompt and $\mathcal{C}$ is the restricted space of admissible solver modifications. These quantities are not solved exactly; they only distinguish improvements achievable through prompt refinement from those requiring changes to the inference procedure.

In practice, PACE uses recent validation improvement from the prompt-evolution tool as a conservative trigger, computed after round t as $\widehat{\Delta U}_P^{(t)} = \widehat{U}(A(P_t, C); \mathcal{V}) - \widehat{U}(A(P_{t-1}, C); \mathcal{V})$. The controller continues prompt evolution while $\widehat{\Delta U}_P^{(t)} > \epsilon$; once $\widehat{\Delta U}_P^{(t)} \leq \epsilon$, prompt refinement is treated as saturated, and admissible control-logic proposals are considered.

3.3.2 Empirical Validation of Solver Updates

Empirical evidence (Table 5). Control-logic edits proposed by SLMs can easily lead to failures. Beyond producing invalid tool calls or non-executable code, SLMs may introduce semantically incorrect solver modifications that preserve syntactic validity but degrade performance. For instance, we observed cases where an SLM rewrote a correct solver into a verification-style procedure that omitted essential task context: the modified solver remained executable, yet caused a clear accuracy regression. Since such edits can pass runtime checks and may appear locally plausible, solver updates should not be accepted based on model self-assessment alone.

PACE addresses this by treating every control-logic edit as a candidate requiring empirical validation. We introduce `action_compare_variants`, which compares two control logic variants on the same validation subset, returning per-variant performance, sample-level differences, and an accept/reject recommendation. Formally, let A_{old} and A_{new} denote the current and candidate agents. Given a validation subset $\mathcal{V} \subset \mathcal{T}$, the candidate is accepted only if

$$\widehat{U}(A_{\text{new}}; \mathcal{V}) > \widehat{U}(A_{\text{old}}; \mathcal{V}) + \delta \quad \text{and} \quad \text{Cost}(A_{\text{new}}) \leq B,$$

where $\widehat{U}(\cdot; \mathcal{V})$ denotes empirical utility on the shared validation subset and $\delta \geq 0$ is the *validation gate threshold*, a minimum improvement margin that filters out noisy or marginal gains. Setting δ=0

accepts any non-regressing candidate; larger values require stronger evidence before committing the update. In practice, we set δ=0.02, and we ensure validation comparison samples a fresh subset from the training pool, varying both size and composition across iterations. This reduces repeated selection on a fixed slice, but does not guarantee generalization or convergence.

4 Experiment

We evaluate PACE on four static benchmarks using three frozen SLM backbones from 4B to 14B parameters, comparing against prompt-only and control-logic-only evolution while analyzing threshold sensitivity and token overhead. We further evaluate PACE on τ-bench as a realistic multi-turn tool-use case study. Additional results and evolved solver listings are provided in the Appendix.

4.1 Setup

In our implementation, B denotes the evolution budget. Each run uses a frozen SLM backbone with at most 14B parameters and is capped at 20 evolution steps. We do not enforce a fixed per-query token budget during evolution; instead, we measure the inference-time cost of the evolved agent, including model calls and generated tokens, and report the resulting accuracy–cost tradeoff (Appendix A.6).

Benchmarks and Evaluation Metrics. We conduct controlled comparisons on four static benchmarks: MMLU [Hendrycks et al., 2020] for knowledge-intensive multiple-choice reasoning, MGSM [Shi et al., 2022] for multilingual math reasoning, HotpotQA [Yang et al., 2018] for multi-hop question answering, and IFEval [Zhou et al., 2023] for verifiable instruction following. We report task-standard accuracy metrics, including letter-match accuracy for MMLU, exact numeric match for MGSM, exact match for HotpotQA, and strict instruction accuracy for IFEval. We further evaluate PACE on τ-bench [Yao et al., 2024] as a realistic multi-turn tool-use case study involving simulated users, domain policies, and API tools. Full dataset splits and evaluation details are in Appendix A.1. Note that the utility U is instantiated by each benchmark's primary metric: accuracy for MMLU, exact match for MGSM and HotpotQA, strict accuracy for IFEval, and task success for τ-bench.

SLM backbones. All experiments use frozen SLMs served locally via vLLM [Kwon et al., 2023]. We study three backbone families ranging from 4B to 14B parameters: 1) QWEN/QWEN3-4B-INSTRUCT-2507 [Yang et al., 2025], 2) QWEN/QWEN3.5-9B, and 3) MINISTRAL-3-14B-INSTRUCT-2512. Each backbone serves both as the inference model inside the solver and as the controller model that proposes control-logic edits. No stronger teacher model is used during evolution.

Baselines. We compare Vanilla (unmodified solver); +PE (prompt edits only); +CE (PACE's edit space and validation gate, enabled immediately without PE); GEPA [Agrawal et al., 2025] and MIPROv2 [Opsahl-Ong et al., 2024] (prompt optimizers); Gödel Agent [Yin et al., 2025] (broader executable self-modification); and ACE [Zhang et al., 2025] (textual context/playbook evolution with a fixed harness). Adaptive methods share the frozen backbone, controlled splits, metrics, and 20-step evolution budget (Appendix A.9). DeepSeek-V3.2-685B provides a frontier reference. Main results report means over ten independent seeded runs; Appendix 8 reports standard deviations.

4.2 Quantitative Results

Table 1 reports performance across three SLM backbones on four controlled benchmarks, with DeepSeek-V3.2 included as a frontier reference. Across all SLM backbones, PACE achieves the best overall performance among the evaluated adaptation methods. The strongest gains are observed on MMLU (+6.1% to +8.6% accuracy) and IFEval (+5.2% to +9.2%), where PACE consistently outperforms both prompt-only (+PE) and control-logic-only (+CE) evolution. Compared with the DeepSeek-V3.2 reference, PACE substantially narrows the vanilla-to-frontier gap. For example, on Qwen3.5-9B, PACE closes 79% of the MMLU gap and 38% of the IFEval gap, while on Ministral-3-14B it closes 50% of the MMLU gap and 67% of the IFEval gap.

Comparison to single-axis evolution. Across backbones, +PE and +CE each capture partial gains, while PACE combines and exceeds them. For example, on Qwen3.5-9B MMLU, +PE reaches 0.832

Table 1: Performance comparison, where CE denotes control logic evolution only, PE denotes prompt evolution only, and $+\%$ denotes relative performance gain. Best result per backbone is **bolded**.

SLM	Method	MMLU	+%	MGSM	+%	HotpotQA	+%	IFEval	+%
	DeepSeek V3.2	0.908	–	0.932	–	0.779	–	0.865	–
Qwen3-4B	Vanilla	0.771	–	0.795	–	0.775	–	0.691	–
	MIPROv2	0.771	+0.0	0.795	+0.0	0.758	-2.2	0.691	+0.0
	GEPA	0.771	+0.0	0.802	+0.9	0.775	+0.0	0.676	-2.2
	ACE	0.800	+3.8	0.808	+1.6	0.775	+0.0	0.711	+2.9
	Gödel Agent	0.797	+3.4	0.812	+2.1	0.772	-0.4	0.705	+2.0
	+CE	0.797	+3.4	0.821	+3.3	0.770	-0.6	0.722	+4.5
	+PE	0.788	+2.2	0.815	+2.5	0.775	+0.0	0.705	+2.0
	+PACE	**0.818**	**+6.1**	**0.831**	**+4.5**	**0.776**	**+0.1**	**0.727**	**+5.2**
Qwen3.5-9B	Vanilla	0.818	–	0.858	–	0.754	–	0.697	–
	MIPROv2	0.797	-2.6	0.861	+0.3	0.739	-2.0	0.697	+0.0
	GEPA	0.832	+1.7	0.878	+2.3	0.741	-1.7	0.704	+1.0
	ACE	0.805	-1.6	0.889	+3.6	0.771	+2.3	0.718	+3.0
	Gödel Agent	0.835	+2.1	0.864	+0.7	0.780	+3.4	0.732	+5.0
	+CE	0.835	+2.1	0.904	+5.4	0.752	-0.3	0.725	+4.0
	+PE	0.832	+1.7	0.872	+1.6	0.789	+4.6	0.718	+3.0
	+PACE	**0.889**	**+8.7**	**0.909**	**+5.9**	**0.803**	**+6.5**	**0.761**	**+9.2**
Ministral-3-14B	Vanilla	0.800	–	0.770	–	0.596	–	0.781	–
	MIPROv2	0.833	+4.1	0.783	+1.7	0.583	-2.2	0.801	+2.6
	GEPA	0.825	+3.1	0.775	+0.6	0.625	+4.9	0.816	+4.5
	ACE	0.818	+2.2	0.785	+1.9	0.605	+1.5	0.788	+0.9
	Gödel Agent	0.797	-0.4	0.825	+7.1	0.659	+10.6	0.810	+3.7
	+CE	0.836	+4.5	0.880	+14.3	0.654	+9.7	0.828	+6.0
	+PE	0.832	+4.0	0.814	+5.7	0.709	+19.0	0.816	+4.5
	+PACE	**0.854**	**+6.7**	**0.908**	**+17.9**	**0.774**	**+29.9**	**0.837**	**+7.2**

and +CE reaches 0.835, whereas PACE achieves 0.889, approaching the DeepSeek-V3.2 reference score of 0.908. On Qwen3.5-9B IFEval, PACE improves by $+9.2\%$ over vanilla, compared with $+3.0\%$ for +PE and $+4.0\%$ for +CE. These results suggest that prompt and control-logic updates address complementary failure modes. The evolution trajectories in Figure 1 further illustrate why both mechanisms are needed: prompt-only evolution saturates, control-logic-first evolution is noisy, and PACE delays structural updates until prompt gains plateau.

Comparison to existing optimizers. MIPROv2, GEPA, and ACE show inconsistent gains and occasional regressions. Gödel Agent is more competitive but trails PACE on most benchmarks. The schedule and gate ablations below test whether coordinating prompt and structural adaptation contributes to this advantage.

Benchmark-specific trends. On MGSM and HotpotQA, +CE already provides large gains for the 9B and 14B backbones, and PACE further improves performance, indicating that multi-step reasoning and retrieval-style tasks benefit strongly from CE such as verification, structured reasoning passes, or retry mechanisms. Notably, PACE with Qwen3.5-9B surpasses the DeepSeek-V3.2 reference on HotpotQA, reaching 0.803 compared with 0.779, while Ministral-3-14B with PACE nearly matches DeepSeek-V3.2 on HotpotQA, reaching 0.774. In contrast, Qwen3-4B shows little improvement on HotpotQA across all methods, suggesting that model capacity can remain the limiting factor even with improved agent design. On Qwen3.5-9B MMLU, the structural update adds 5.7 points beyond PE, illustrating the remaining headroom after prompt refinement.

Statistical robustness. Results average ten independent evolution runs. For Qwen3.5-9B, Welch 95% confidence intervals for PACE's gains over Vanilla, +PE, and +CE are respectively $[6.5, 7.7]$, $[5.0, 6.4]$, and $[4.0, 6.8]$ percentage points on MMLU, and $[5.4, 7.4]$, $[3.2, 5.4]$, and $[1.4, 5.8]$ on IFEval. All exclude zero (Appendix A.9).

Cost analysis. Estimated one-time evolution costs for Qwen3.5-9B are 2.1M/3.0M/2.5M/3.9M tokens on MMLU/IFEval/MGSM/HotpotQA. PE uses 1.1–1.6× baseline deployment tokens, whereas PACE uses 1.6–5.2× (Appendix A.6). Measured MMLU latency increases to 1.30× while accuracy rises from .818 to .889 (Table 2). Parallel voting samples share a batched serving request; conditional verification raises the average to 1.2 requests/query. PACE offers an accuracy–cost tradeoff, without equal-cost dominance or universal superiority to fine-tuning.

Table 2: Qwen3.5-9B MMLU serving costs under the same vLLM/A100 configuration. Latency is measured relative to Vanilla; token counts are estimates.

Method	Requests/query	Tokens/query	Relative latency	Accuracy
Vanilla	1.0	470	1.00×	.818
+PE	1.0	570	1.05×	.832
+CE	1.2	1,610	1.25×	.835
PACE	1.2	1,710	1.30×	.889

Table 3: Performance on τ-bench[3]: we leverage Claude Sonnet 4.5 as the stochastic user simulator across Retail (115 tasks) and Airline (50 tasks) domains. Vanilla, +PE, and +PACE are evaluated in both domains; additional Qwen3.5-9B baselines cover Retail only (–: not evaluated). pass^k denotes the fraction of tasks solved in *all* k independent trials. Best result per backbone is **bolded**.

		Retail (115 tasks)				**Airline** (50 tasks)			
SLM	**Method**	pass^1	pass^2	pass^3	pass^4	pass^1	pass^2	pass^3	pass^4
	Sonnet 3.5 [Yao et al., 2024]	.692	.576	.509	.462	.460	.326	.263	.225
	GPT-4o [Yao et al., 2024]	.604	.491	.430	.383	.420	.273	.220	.200
	Vanilla	.465	.332	.261	.217	.265	**.160**	**.130**	**.120**
Qwen3-4B	+PE	.474	.341	.278	.226	.235	.097	.050	.040
	+PACE	**.513**	**.378**	**.307**	**.270**	**.275**	.153	.100	.060
	Vanilla	.791	.706	.659	.626	.535	.417	.355	.320
	+CoT	.798	.703	.639	.591	–	–	–	–
	+Few-shot	.815	.736	.685	.652	–	–	–	–
Qwen3.5-9B	+GEPA	.791	.706	.659	.626	–	–	–	–
	+Gödel Agent	.807	.694	.662	.596	–	–	–	–
	+PE	.785	.704	.652	.617	.550	.450	.390	.340
	+PACE	**.817**	**.745**	**.698**	**.661**	**.575**	**.457**	**.395**	**.360**

τ-bench case study. Table 3 evaluates PACE on realistic multi-turn tool-use tasks, where the agent must maintain dialogue state, follow domain policies, and issue valid API calls while interacting with a stochastic user simulator. Prompt-only evolution is unstable in this setting: +PE improves Qwen3-4B Retail but degrades Qwen3.5-9B Retail (0.785 vs. 0.791) and collapses on Qwen3-4B Airline (pass^4: 0.040 vs. 0.120). On Qwen3.5-9B Retail, PACE leads all added baselines at every pass^k, including few-shot prompting (.817 vs. .815 at $k = 1$; .661 vs. .652 at $k = 4$). CoT slightly improves single-trial success but degrades repeated-trial consistency. Qwen3-4B Airline remains a limitation: PACE improves pass^1 but trails Vanilla for $k \geq 2$. Notably, Qwen3.5-9B + PACE surpasses both the Sonnet 3.5 and GPT-4o references [Yao et al., 2024] on Retail and Airline across all pass^k levels.

4.3 Ablation Study and Parameter Sensitivity Analysis

We ablate two core design choices in PACE using Qwen3.5-9B on MMLU and IFEval: (i) the credit-assignment threshold ε that gates the transition from prompt optimization to structural exploration, and (ii) the validation gate threshold δ that filters proposed solver edits before they are committed. Table 4 reports test accuracy under each setting. For the ε sweep (left), ε=0 disables control logic evolution entirely (prompt-only), while ε=1.0 bypasses saturation detection and unlocks structure edits after the first prompt round. For the δ sweep (right), δ=-1 disables the gate (any candidate is accepted), while δ=0.05 requires the candidate to exceed the current solver by at least 5%.

Credit-assignment threshold ε. In Table 4, both extremes hurt: ε=0 (prompt-only) never enters structural search, while ε=1.0 unlocks structure edits before the prompt is sufficiently refined. The default $\epsilon = 0.01$ achieves the best accuracy on both benchmarks, improving over the prompt-only setting by $\sim 5\%$ on both benchmarks. These gains indicate CE can provide substantial improvements beyond PE alone. The $\varepsilon = 1.0$ setting approximates a naïve combination that allows control-logic edits before prompt refinement has saturated. Its lower performance, especially on IFEval, shows combining prompt and control-logic search is insufficient; the transition policy matters.

Validation gate threshold δ. Disabling the gate (δ=-1) degrades IFEval by 3.8%, as harmful edits pass unchecked. On MMLU the effect is smaller, since MMLU's multiple-choice format leaves less

[3]We omit the CE-only baseline because multi-turn dialogue performance is highly sensitive to prompt quality; applying structural evolution from an unoptimized prompt yields unstable trajectories that do not constitute a meaningful ablation.

Table 4: Ablation on credit-assignment threshold ε (left) and control logic validation gate δ (right) with Qwen 3.5-9B as the base model. Shaded columns mark the proposed defaults (ε=0.01, δ=0.02).

	Credit-Assignment Threshold (ε)						Validation Gate Threshold (δ)			
Benchmark	0	.005	.01	.02	.05	1.0	−1	0	.02	.05
MMLU	83.2	87.9	**88.9**	88.6	88.3	88.4	88.3	88.8	**88.9**	88.9
IFEval	71.8	73.4	**76.1**	74.8	73.1	72.5	72.3	74.6	**76.1**	75.2

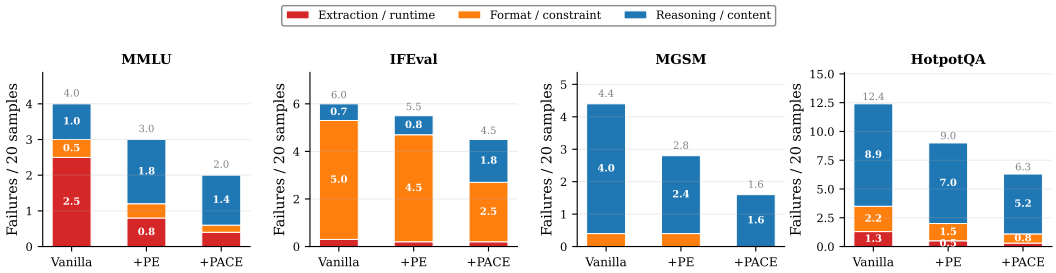

Figure 3: Failure mode distribution per 20-sample mini-batch across evolution phases on all benchmarks with Qwen3.5-9B. PE resolves extraction and format errors; residual failures after +PE are dominated by reasoning or hard constraint violations, which CE then targets.

room for structurally broken solvers to diverge. Overly strict gating (δ=0.05) also underperforms the default, as it blocks beneficial edits that show moderate but genuine improvement on the small validation subset. The default δ=0.02 strikes the best balance, filtering noisy candidates while admitting meaningful structural gains.

4.4 Failure Mode Shift Across Evolution Phases

The two-timescale design rests on the premise that prompt and structural interventions resolve *different* failure modes. To validate this, we classify every failure on a 20-sample mini-batch into three categories—*extraction/runtime* (malformed output, parse errors), *format/constraint* (valid output violating task requirements), and *reasoning/content* (well-formed but incorrect)—and track how the distribution shifts from Vanilla through +PE to +PACE.

Fig. 3 further shows PE reduces extraction/format errors, while the residual failures after +PE are increasingly dominated by reasoning/content or hard-constraint errors. This shift supports the saturation trigger: once prompt-level gains plateau, the remaining errors are more likely to require changes to the solver's inference procedure, such as voting, verification, or task-specific extraction.

4.5 Control-Logic Proposal Filtering

The frozen SLM generates useful but noisy proposals: only 19 of 87 are accepted (Table 5). Voting and retry/repair account for 14 of 19 accepted edits. The 68 rejections comprise validation regression (20), budget violations (18), insufficient gain (15), and runtime failure (15). Parsing/normalization is most often proposed and rejected; 100% acceptance for retry/repair and routing reflects small samples. A strict JSON MMLU parser (.83→.00) and an IFEval boilerplate post-processor (.58→.42) illustrate harmful executable edits blocked by validation (Appendix A.9).

4.6 Larger-Model Scaling Check

PACE's mechanisms are model-size agnostic, but its empirical benefit is scoped to frozen SLMs. On DeepSeek-V4-Flash (284B total, 13B activated parameters), no adaptation method improves every task (Table 6). Full PACE lowers mean performance by 0.3 percentage points. This is consistent with reduced adaptation headroom for a stronger backbone; a single scaling check does not establish a general relationship between size and improvement.

Table 5: Control-logic proposal audit across three PACE runs on MMLU and IFEval with Qwen3.5-9B. Rejected proposals include non-executable candidates and executable edits that were not committed.

Edit type	Proposed	Accepted	Rejected	Acceptance rate
Parsing / normalization	36	2	34	5.6%
Validation	14	0	14	0.0%
Retry / repair	6	6	0	100.0%
Self-consistency / voting	22	8	14	36.4%
Verification	8	2	6	25.0%
Routing / fallback	1	1	0	100.0%
Total	87	19	68	21.8%

Table 6: Larger-model scaling check on DeepSeek-V4-Flash. HotpotQA uses F1; Δ is the change in the rounded four-task mean, in percentage points, relative to Vanilla.

Method	MMLU	MGSM	HotpotQA F1	IFEval	Mean	Δ (pp)
Vanilla	.901	.911	.797	.854	.866	–
Gödel Agent	.890	.915	.797	.861	.866	+0.0
MIPROv2	.897	.914	.794	.858	.866	+0.0
GEPA	.896	.915	.805	.865	.870	+0.4
ACE	.890	.916	.795	.865	.867	+0.1
PACE CE-only	.890	.912	.802	.839	.861	-0.5
PACE PE-only	.890	.916	.777	.847	.858	-0.8
PACE (full)	.886	.915	.798	.851	.863	-0.3

5 Limitations

The static benchmarks support controlled within-protocol comparisons; the agentic evaluation covers only τ-bench, and no multimodal tasks are evaluated. Gains on frozen SLMs do not imply gains on larger models. Evolution and deployment both consume additional compute, and latency depends on serving hardware and batching. Validation supports empirical filtering rather than a convergence, generalization, or safety guarantee. Repeated validation can introduce selection bias. Bounded interfaces and rejection of malformed or regressing proposals are operational safeguards; high-stakes and adversarial deployment remain untested. Accepted-edit frequencies describe search behavior, not an additive causal attribution of performance gains.

6 Conclusion

We introduced PACE, a two-timescale agentic framework for frozen SLM self-evolution that coordinates fast prompt refinement with less frequent, constrained control-logic updates. By separating proposal from validation, PACE lets the SLM generate candidate edits while committing only those that improve held-out validation performance under a resource budget. Across four controlled benchmarks and three SLM backbones (4B–14B), PACE leads the evaluated adaptation methods, at an estimated one-time evolution cost of 2.1–3.9M tokens. On Qwen3.5-9B, MMLU and IFEval improve by 7.1 and 6.4 percentage points over Vanilla. Expanded τ-bench comparisons support its utility in multi-turn interaction; the larger-model check and serving costs delimit that benefit. More broadly, PACE shows that the novelty is not in any single final solver pattern, but in enabling a frozen SLM to autonomously discover, select, and validate task-appropriate inference strategies; the resulting accepted and rejected evolution trajectories also provide preference- or reward-labeled data for training future evolution controllers.

References

Lei Wang, Chen Ma, Xueyang Feng, Zeyu Zhang, Hao Yang, Jingsen Zhang, Zhiyuan Chen, Jiakai Tang, Xu Chen, Yankai Lin, et al. A survey on large language model based autonomous agents. *Frontiers of Computer Science*, 18(6):186345, 2024a.

Lakshya A Agrawal, Shangyin Tan, Dilara Soylu, Noah Ziems, Rishi Khare, Krista Opsahl-Ong, Arnav Singhvi, Herumb Shandilya, Michael J Ryan, Meng Jiang, et al. Gepa: Reflective prompt evolution can outperform reinforcement learning. *arXiv preprint arXiv:2507.19457*, 2025.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. Optimizing instructions and demonstrations for multi-stage language model programs. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 9340–9366, 2024.

Qizheng Zhang, Changran Hu, Shubhangi Upasani, Boyuan Ma, Fenglu Hong, Vamsidhar Kamanuru, Jay Rainton, Chen Wu, Mengmeng Ji, Hanchen Li, et al. Agentic context engineering: Evolving contexts for self-improving language models. *arXiv preprint arXiv:2510.04618*, 2025.

Zhengwei Tao, Ting-En Lin, Xiancai Chen, Hangyu Li, Yuchuan Wu, Yongbin Li, Zhi Jin, Fei Huang, Dacheng Tao, and Jingren Zhou. A survey on self-evolution of large language models. *arXiv preprint arXiv:2404.14387*, 2024.

Jinyuan Fang, Yanwen Peng, Xi Zhang, Yingxu Wang, Xinhao Yi, Guibin Zhang, Yi Xu, Bin Wu, Siwei Liu, Zihao 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.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in neural information processing systems*, 36:8634–8652, 2023.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc V Le, Ed H. Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. Self-consistency improves chain of thought reasoning in language models. In *The Eleventh International Conference on Learning Representations*, 2023.

Xingyao Wang, Yangyi Chen, Lifan Yuan, Yizhe Zhang, Yunzhu Li, Hao Peng, and Heng Ji. Executable code actions elicit better llm agents. In *Forty-first International Conference on Machine Learning*, 2024b.

Jürgen Schmidhuber. Gödel machines: self-referential universal problem solvers making provably optimal self-improvements. *arXiv preprint cs/0309048*, 2003.

Xunjian Yin, Xinyi Wang, Liangming Pan, Li Lin, Xiaojun Wan, and William Yang Wang. Gödel agent: A self-referential agent framework for recursively self-improvement. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 27890–27913, 2025.

Han Zhou, Xingchen Wan, Ruoxi Sun, Hamid Palangi, Shariq Iqbal, Ivan Vulić, Anna Korhonen, and Sercan Ö Arık. Multi-agent design: Optimizing agents with better prompts and topologies. *arXiv preprint arXiv:2502.02533*, 2025.

Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *Proceedings of the 2023 conference on empirical methods in natural language processing*, pages 7957–7968, 2023.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Freda Shi, Mirac Suzgun, Markus Freitag, Xuezhi Wang, Suraj Srivats, Soroush Vosoughi, Hyung Won Chung, Yi Tay, Sebastian Ruder, Denny Zhou, et al. Language models are multilingual chain-of-thought reasoners. *arXiv preprint arXiv:2210.03057*, 2022.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher 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.

Jeffrey Zhou, Tianjian Lu, Swaroop Mishra, Siddhartha Brahma, Sujoy Basu, Yi Luan, Denny Zhou, and Le Hou. Instruction-following evaluation for large language models. *arXiv preprint arXiv:2311.07911*, 2023.

Shunyu Yao, Noah Shinn, Pedram Razavi, and Karthik Narasimhan. τ-bench: A benchmark for tool-agent-user interaction in real-world domains, 2024. URL `https://arxiv.org/abs/2406.12045`.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

A Appendix

A.1 Benchmark Details and Data Splits

This section documents the four benchmarks used throughout the paper (MMLU, IFEval, HotpotQA, and MGSM) and specifies the data splits, program structure, and evaluation metrics shared by the Vanilla, +PE, and +PACE conditions as well as the DSPy baselines (GEPA, MIPROv2) reported in the main results.

Common protocol. Every benchmark is partitioned into three disjoint, seed-shuffled splits. The *train* split is used by the prompt evolution optimizer to sample minibatches and by DSPy baselines to bootstrap demonstrations. The *validation* split is used for comparisons of candidate solvers or prompts (including the PACE credit-assignment gate). The *test* split is held out and used only for the final reported numbers. The documented seed-0 split construction uses the `auto="medium"` budget preset for DSPy optimizers, and identical splits across Vanilla, +PE, +PACE, GEPA, and MIPROv2 so that differences reflect the method rather than the data.

Each task is implemented as a single-module DSPy `ChainOfThought` predictor. The signature exposes an input schema, a `reasoning` field, and a task-specific output field; only the signature instruction is subject to optimization. At inference time the model is served via an OpenAI-compatible interface backed by vLLM, with a round-robin dispatcher sharding requests across replicas.

Table 7: Controlled data splits used for repeated evolution experiments. All methods use the same train/test subsets; validation subsets are dynamically sampled by the agent during evolution. Results should be interpreted as within-protocol comparisons rather than standard leaderboard scores.

Task	Train	Val	Test	Test source	Primary metric
MMLU	200	dynamic	800	official test subset	accuracy
IFEval	200	dynamic	141	held-out from official prompts	strict instruction accuracy
HotpotQA	200	dynamic	500	official validation subset	exact match (+ F1)
MGSM	200	dynamic	800	pooled multilingual subset	exact numeric match

Because evolution requires repeated evaluation, we use fixed controlled subsets for training and testing rather than full benchmark test suites; the same splits are shared across all methods. For

IFEval, whose official release consists of a single set of 541 prompts without a designated train/test partition, we randomly split the full prompt set into disjoint train and test subsets using a fixed seed.

Validation subsets are not fixed: at each evaluation step, the agent dynamically samples a fresh subset from the training pool, varying both size and composition across iterations to reduce overfitting to any single validation slice. This design reflects a practical deployment constraint where repeated evaluation on an identical held-out set can lead to implicit selection bias in the evolution trajectory.

Following standard practice in iterative prompt/agent optimization [Opsahl-Ong et al., 2024, Pryzant et al., 2023], we evaluate on fixed controlled subsets to enable repeated evaluation during evolution while keeping compute tractable. All methods share identical splits, ensuring fair within-protocol comparison.

MMLU (multiple-choice knowledge). MMLU is a closed-book multiple-choice benchmark covering 57 subjects with four answer choices per question.

- **Data.** The full dataset is loaded from Kaggle[4]. We sampled 800 cases as held-out test set.
- **Program.** `ChainOfThought` over a signature with input `question` and outputs `reasoning` and `answer`, where `answer` is constrained to one letter in $\{A, B, C, D\}$.
- **Metric.** We report letter-match accuracy. The first A/B/C/D token is extracted from the model output case-insensitively, and the prediction is counted correct iff it matches the gold letter.

IFEval (verifiable instruction following). IFEval measures whether a model satisfies verifiable formatting and content constraints attached to each prompt.

- **Data.** We shuffle the available IFEval prompts and use 200 examples for training and 100 examples for validation. A disjoint 141-example subset from the official test set is reserved for final evaluation.
- **Program.** `ChainOfThought` over a signature with input `prompt` and outputs `reasoning` and `response`. The `response` field is scored by the instruction checkers.
- **Metrics.** We use the IFEval instruction checkers for length, case, keyword, format, and punctuation constraints. Each response is scored by *strict accuracy*, which equals 1.0 only if all attached instructions are satisfied, and *loose accuracy*, the fraction of satisfied instructions. Strict accuracy is used as the optimization objective; failure feedback lists violated instruction identifiers and the loose score.

HotpotQA (multi-hop QA, distractor setting). HotpotQA Yang et al. [2018] requires combining evidence from multiple Wikipedia paragraphs to answer a question. In the distractor setting, each example includes ten paragraphs, consisting of two gold paragraphs and eight distractors.

- **Data.** We use the HuggingFace `hotpot_qa` distractor split. After shuffling the training split, we use 200 examples for training and 100 for validation. Final evaluation uses the first 500 examples from the official validation split. Paragraphs are concatenated into a single context string with title markers and sentence text.
- **Program.** `ChainOfThought` over a signature with inputs `context` and `question` and outputs `reasoning` and `answer`, where `answer` is a short span.
- **Metrics.** We apply SQuAD-style normalization, including lowercasing, article removal, punctuation removal, and whitespace normalization. Exact Match is used as the optimizer objective, and token-level F1 is also reported for analysis.

MGSM (multilingual grade-school math). MGSM is a multilingual grade-school math benchmark with numeric answers, covering 11 languages: Bengali, German, English, Spanish, French, Japanese, Russian, Swahili, Telugu, Thai, and Chinese.

- **Data.** For each language, we load `mgsm_{lang}.tsv`, prepend a language-specific instruction prefix, pool examples across languages, shuffle with seed 0, and split the pooled data into 200 training, 100 validation, and 800 test examples.
- **Program.** `ChainOfThought` over a signature with input `question` and outputs `reasoning` and `answer`, where `answer` is numeric.

[4] https://www.kaggle.com/datasets/open-benchmarks/mmlu-massive-multitask-language-understanding/data

- **Metric.** We report exact numeric match. The model output is normalized by extracting the first numeric token, removing thousands separators, and trimming trailing zeros and decimal points; the prediction is correct iff the normalized value matches the gold answer.
- **Inference note.** Non-English problems occasionally produce long repetitive generations, so we set `max_tokens` to 8000 and use task-generation temperature 0.6 for MGSM. Parse failures caused by truncation are counted as incorrect rather than aborting evaluation.

τ-bench (multi-turn tool-augmented dialogue). τ-bench [Yao et al., 2024] is a realistic agent benchmark that evaluates LLM agents in dynamic, multi-turn customer-service conversations requiring policy-compliant tool use. The agent interacts with a stochastic LLM-simulated user who has a hidden intent (e.g., exchange an item, cancel an order), and must resolve the request by calling domain-specific API tools (e.g., `get_order_details`, `exchange_delivered_order_items`) while adhering to a detailed policy document (the "wiki").

- **Data.** We evaluate on two domains: *Retail* (115 tasks involving order management, returns, exchanges, and address modifications) and *Airline* (50 tasks involving flight changes, cancellations, and baggage policies). Vanilla, +PE, and +PACE are evaluated with Qwen3-4B and Qwen3.5-9B in both domains; the additional Qwen3.5-9B CoT, few-shot, GEPA, and Gödel Agent baselines are evaluated only on Retail. Each task specifies a user persona with a hidden intent and a gold action sequence; the environment is fully stateful with a database backend.
- **Program.** The agent runs a multi-turn loop: at each step it receives the user's message (or tool result), generates either a tool call or a natural-language response, and the environment advances accordingly. The system prompt consists of the domain policy wiki concatenated with any evolved role description and requirements. The agent has access to 10–15 domain-specific tools per environment. Maximum conversation length is 30 steps (tool calls + responses).
- **Metric.** Each task is evaluated over $k{=}4$ independent trials with the stochastic user simulator. pass^k denotes the fraction of tasks where the agent succeeds (reward $= 1.0$) in *all* k trials, computed via the combinatorial estimator $\text{pass}_i^k = \binom{c_i}{k} / \binom{n}{k}$ where c_i is the number of successful trials for task i. This metric captures both accuracy and behavioral consistency under user variability.
- **User simulator.** We use Claude Sonnet 4.5 (`claude-sonnet-4-5-20250929`) as the user simulator for all reported results. The simulator follows the task's hidden intent, responds naturally to the agent, and terminates the conversation (`###STOP###`) if the agent becomes unresponsive or completes the task.
- **Inference note.** The agent model (Qwen3.5-9B) is served locally via vLLM on 2 A100 GPUs with temperature 0.0. During evolution, validation uses single-trial evaluation on all tasks for efficiency; the final reported metrics use 4-trial evaluation. Each full 4-trial evaluation run takes approximately 8 hours per domain.

DSPy baselines. The GEPA and MIPROv2 baselines reported in the main tables use the splits above, the same `ChainOfThought` program for each task, and identical evaluation metrics. Both optimizers are run with `auto="medium"` and seed 0. GEPA additionally uses a reflection language model at temperature 1.0; MIPROv2 uses its default grounded proposer. To keep comparisons fair with PACE, DSPy's `max_errors` threshold is set to the size of the evaluation set so that sporadic parse failures on pathological examples are counted as incorrect rather than aborting the evaluation.

Inference infrastructure. All experiments run on a single node with eight vLLM replicas of the target SLM, each bound to one GPU. When four experiments share the cluster we allocate two replicas per task and use 24 evaluation threads; when a single task has exclusive use of the cluster we use all eight replicas with 48 threads. Task-generation temperature is 0.2 for MMLU, IFEval, and HotpotQA and 0.6 for MGSM; optimizer-side reflection temperature is fixed at 1.0.

A.2 Experiment Settings

Agent Tools Description. The following list covers existing tools for agent control logic optimization. Note that `action_optimize_prompt_on_task` is the prompt optimization that would run by default in each iteration.

- `action_display_analysis`: Summarize the latest evaluation failures into a structured failure taxonomy (extraction/runtime, format/constraint, reasoning/content) and produce actionable evolution guidance with ranked failure modes and fix recommendations.

Table 8: Performance comparison across SLM backbones. CE denotes control-logic evolution only, PE denotes prompt evolution only, and PACE denotes the full two-timescale framework. Results are reported as mean ± standard deviation over multiple runs. Best result per backbone is **bolded**.

SLM	Method	MMLU	MGSM	HotpotQA	IFEval
Qwen3-4B	Vanilla	0.771 ± 0.005	0.795 ± 0.006	0.775 ± 0.012	0.691 ± 0.008
	MIPROv2	0.771 ± 0.005	0.795 ± 0.007	0.758 ± 0.014	0.691 ± 0.009
	GEPA	0.771 ± 0.005	0.802 ± 0.007	0.775 ± 0.013	0.676 ± 0.010
	ACE	0.800 ± 0.006	0.808 ± 0.008	0.775 ± 0.015	0.711 ± 0.011
	Gödel Agent	0.797 ± 0.015	0.812 ± 0.022	0.772 ± 0.048	0.705 ± 0.055
	+CE	0.797 ± 0.021	0.821 ± 0.024	0.770 ± 0.038	0.722 ± 0.031
	+PE	0.788 ± 0.006	0.815 ± 0.008	0.775 ± 0.014	0.705 ± 0.010
	+PACE	$\mathbf{0.818 \pm 0.007}$	$\mathbf{0.831 \pm 0.009}$	$\mathbf{0.776 \pm 0.015}$	$\mathbf{0.727 \pm 0.011}$
Qwen3.5-9B	Vanilla	0.818 ± 0.005	0.858 ± 0.009	0.754 ± 0.017	0.697 ± 0.009
	MIPROv2	0.797 ± 0.006	0.861 ± 0.010	0.739 ± 0.019	0.697 ± 0.010
	GEPA	0.832 ± 0.006	0.878 ± 0.011	0.741 ± 0.018	0.704 ± 0.010
	ACE	0.805 ± 0.007	0.889 ± 0.011	0.771 ± 0.020	0.718 ± 0.011
	Gödel Agent	0.835 ± 0.024	0.864 ± 0.028	0.780 ± 0.043	0.732 ± 0.038
	+CE	0.835 ± 0.019	0.904 ± 0.025	0.752 ± 0.036	0.725 ± 0.029
	+PE	0.832 ± 0.007	0.872 ± 0.011	0.789 ± 0.018	0.718 ± 0.011
	+PACE	$\mathbf{0.889 \pm 0.007}$	$\mathbf{0.909 \pm 0.012}$	$\mathbf{0.803 \pm 0.021}$	$\mathbf{0.761 \pm 0.012}$
Ministral-14B	Vanilla	0.800 ± 0.006	0.770 ± 0.008	0.596 ± 0.016	0.781 ± 0.009
	MIPROv2	0.833 ± 0.007	0.783 ± 0.009	0.583 ± 0.018	0.801 ± 0.010
	GEPA	0.825 ± 0.006	0.775 ± 0.010	0.625 ± 0.019	0.816 ± 0.011
	ACE	0.818 ± 0.007	0.785 ± 0.011	0.605 ± 0.020	0.788 ± 0.012
	Gödel Agent	0.797 ± 0.027	0.825 ± 0.031	0.659 ± 0.045	0.810 ± 0.041
	+CE	0.836 ± 0.022	0.880 ± 0.026	0.654 ± 0.035	0.828 ± 0.028
	+PE	0.832 ± 0.007	0.814 ± 0.010	0.709 ± 0.022	0.816 ± 0.011
	+PACE	$\mathbf{0.854 \pm 0.007}$	$\mathbf{0.908 \pm 0.011}$	$\mathbf{0.774 \pm 0.020}$	$\mathbf{0.837 \pm 0.012}$

- `action_read_logic`: Read the source code of a specified function, method, or class within a given module.
- `action_adjust_logic`: Modify, add, or delete the source code of a specified function, method, or class within a given module to improve task-solving ability.
- `action_run_code`: Execute Python or shell code and capture the output, errors, and return value.
- `action_call_json_format_llm`: Call an external LLM for assistance with gathering insights, refining strategies, correcting errors, and solving complex problems, returning the response in JSON format.
- `action_select_examples`: Select a representative subset of task examples (diverse, random, or head strategy) for future train, valid, or evaluate calls.
- `action_compare_variants`: Run a candidate comparison of two solver or prompt variants on the same sampled validation subset to verify whether a change improves accuracy.
- `action_optimize_prompt_on_task`: Run the prompt evolution optimizer, evolving role, requirements, and temperature using minibatch feedback.
- `action_get_evolution_credit`: Return the current evolution credit assignment state, including ΔU_P, ΔU_C, prompt saturation status, and thresholds.
- `action_evaluate_on_task`: Evaluate the current solver on the goal task samples and return evaluation feedback including accuracy and per-sample details.

A.3 Complete Experiment Results for Table 8

Table 8 reports mean ± standard deviation over ten independent evolution runs. PACE remains variable across runs because the controller can discover different solver strategies. For Qwen3.5-9B, the PACE–CE mean difference is 5.4 percentage points on MMLU (.889 versus .835), and the PACE–PE difference is 4.3 points on IFEval (.761 versus .718). Run-level Welch intervals for these comparisons are reported in Appendix A.9. Qwen3-4B HotpotQA remains a narrow cluster with substantial overlap among run-level standard deviations.

A.4 C ontrol Logic Edit Space

Prompt evolution (+PE) and the control logic evolution of PACE share a common interface, `action_adjust_logic`, but the categories of change they are allowed to make on top of the current solver differ qualitatively. This subsection enumerates the concrete edit categories that the controller can emit, specifies whether each is free-form Python or constrained to a template, and lists the safety checks that gate every structural update before it is committed.

Interface. Every structural edit is a single call of the form `action_adjust_logic(module_name, target_name, new_code, target_type, operation)` where `operation`∈{`modify`, `add`, `delete`} and `target_type`∈{`function`, `class`}. By convention the principal target is the solver function (`agent_module.solver`); the same interface can edit or introduce helper functions and classes within `agent_module`. The prompt mutation interface (`action_optimize_prompt_on_task`) is template-based and operates only on the fields of the solver's `prompt_config`—role, requirements, temperature, and response format—without touching control flow.

Edit categories. Table 9 lists the edit categories the agent actually uses across the four benchmarks, together with the target, whether they are free-form or template-constrained, and the safety checks that apply. Templates reflect recurring patterns in what the controller discovers, but are not hard-coded: the agent writes arbitrary Python that conforms to the solver contract (`solver(agent, task)`→`dict`).

Safety constraints. Every structural update is gated by a sequence of checks before it is applied to `agent_module`:

- **Prompt-saturation gate.** Structural updates are blocked unless $\Delta U_P < \varepsilon$ on the PE credit state, i.e. the prompt optimizer has exhausted its gains. This enforces the PE→CE ordering described in the main body.
- **Immutable targets.** The LLM call primitives `Agent.action_call_llm` and `Agent.action_call_json_format_llm` cannot be edited or deleted. Any attempt raises `ValueError` and the edit is discarded.
- **Blacklisted primitives.** The solver may not contain `time.sleep`, and no target may `import logging` or `from logging ...`. These would break the evaluation harness' latency accounting and log capture.
- **Syntax compilation.** Every `modify` or `add` is parsed with `compile(...)`; a `SyntaxError` surfaces as a structured error message (line, column, offending text) and the edit is rejected.
- **validation gate (solver only).** Edits that modify the `solver` function are run through a cheap candidate comparison (`_compare_variant_metrics`) against the current solver on 12 deterministic validation samples. If $\Delta_{\text{AB}} \leq \tau_{\text{AB}}$ (`agent.ab_gate_delta_threshold`) the edit is rejected and the candidate code is not installed.
- **Post-commit structural eval.** When an edit passes the validation gate and is committed, the agent records the current eval accuracy as `structure_eval_baseline` and marks a pending full evaluation. The next `action_evaluate_on_task` computes ΔU_C, which is fed back into the credit mechanism (Section 4.4) and can trigger rollback if the full-evaluation delta is negative.

Implications. The edit space is deliberately asymmetric: prompt fields are small, template-constrained, and cheap to revert, while structural edits are free-form Python but guarded by a prompt-saturation gate, a validation gate, and a set of immutable primitives. This is what lets a frozen 9B model act as its own architect—free-form enough to discover self-consistency on MMLU or the generate–score–repair loop on IFEval (Figures 4–5), while the gates prevent the model's occasional faulty edits from regressing the system.

A.5 Algorithm Walkthrough

Algorithm 1 summarizes the full PACE procedure. Starting from an initial agent, PACE first performs prompt evolution while holding the control logic fixed. Prompt candidates are generated through mutation, reflection, and crossover, then selected using validation utility and resource cost. If the marginal validation improvement from prompt evolution remains above the saturation threshold ϵ, PACE continues refining prompts. Once prompt gains fall below ϵ, the framework activates constrained structural evolution. Structural candidates are proposed only within a predefined safe-to-edit search space and are accepted only if they improve over the current agent by at least the

Table 9: Allowed edit categories in the PACE CE phase. All edits are expressed as calls to `action_adjust_logic`. *Free-form* means the agent writes arbitrary Python satisfying the solver contract; *template-constrained* means the edit is restricted to a fixed schema (prompt fields, temperature, or a hyperparameter dict).

Category	Representative example	Form	Safety constraints
Prompt field update	Change `role`, `requirements`, `temperature`, or `response_format` in `prompt_config`	Template	Schema-validated; no code execution
Sampling-policy change	Replace a single-call decode with n-way sampling at an altered temperature	Free-form	Syntax check; validation gate
Aggregation / selection	Add majority vote (MMLU), score-then-argmax (IFEval), or span-overlap selector (HotpotQA)	Free-form	Syntax check; validation gate
Early-exit gate	Return immediately on $\geq 2/3$ consensus or $s{=}1.0$ constraint pass	Free-form	Syntax check; validation gate
Self-verification step	Add a confirm-or-correct call at $T{=}0$ on the top candidate (MMLU)	Free-form	Syntax check; validation gate
Targeted repair / rewrite	Feed best candidate back as assistant turn and ask for a constraint-satisfying rewrite (IFEval)	Free-form	Syntax check; validation gate; acceptance gate ($s_{\text{repair}} \geq s_{\text{best}}$)
Output normalization	Strip articles, trailing %, commas, or decorations from the answer string	Free-form	Syntax check; validation gate
Helper function / class	Introduce `self_check_constraints`, `extract_answer`, or `normalize_numeric`	Free-form	Must not modify `action_call_llm` or `action_call_json_format_llm`
Hyperparameter dict	Edit temperature bounds, sample count n, retry count, or early-exit threshold	Template	Schema-validated; validation gate
Deletion	Remove an unused helper or a branch that proved harmful	Free-form	Target must not be the solver, `action_call_llm`, or `action_call_json_format_llm`

validation margin δ on the same held-out validation subset without violating the resource budget. After each accepted structural update, PACE returns to prompt evolution, allowing the prompt to adapt to the new control logic.

A.6 Token Usage Analysis

A defining constraint of the SLM setting is that every token is generated locally on limited hardware. Unlike API-based systems where cost is monetary, here cost is *latency and throughput*: each additional LLM call occupies a GPU that could serve another request. We therefore analyze the token budget of both the evolution process and the evolved solvers (inference time) to characterize the practical overhead of self-evolution under resource constraints.[5]

Inference-time token cost. Table 10 decomposes the per-query token budget across solver variants, and two patterns stand out:

[5]Token counts in this section are estimates derived from code-level analysis of prompt templates, solver call patterns, and observed output lengths. The current implementation tracks cost via `len(response_text)` (character count) rather than API-level `usage.prompt_tokens` / `completion_tokens`. Accuracy figures are from real evaluation runs on the respective test splits (800 samples for MMLU, 141 for IFEval).

Algorithm 1 PACE: Prompt And Control Logic Evolution

Require: Frozen SLM M_θ; initial prompt artifacts P_0; initial control logic C_0; training set $\mathcal{D}_{train}$; validation set $\mathcal{D}_{val}$; resource budget B; prompt saturation threshold ϵ; validation gate acceptance margin δ; maximum CE steps K; maximum PE steps L

Ensure: Evolved agent $A^* = (P^*, C^*)$

Initialize $P \leftarrow P_0$, $C \leftarrow C_0$
Evaluate initial agent $A = (P, C)$ on $\mathcal{D}_{val}$ to obtain U_{best}
for $k = 1$ to K **do**
▷ **Prompt evolution under fixed control logic**
  $P_{\text{old}} \leftarrow P$
  for $\ell = 1$ to L **do**
    Generate prompt candidates $\mathcal{P}_{cand}$ using handcrafted mutations, failure-guided reflection, and crossover
    Evaluate each $P' \in \mathcal{P}_{cand}$ with fixed C on minibatches from $\mathcal{D}_{train}$
    Update the prompt Pareto front according to validation utility and resource cost
    Select the best prompt P from the Pareto front using $\mathcal{D}_{val}$
    if $Cost(P, C) > B$ **then**
      Reject P and restore the best feasible prompt on the Pareto front
    end if
  end for
  Compute recent prompt improvement $\Delta U_P \leftarrow U(P, C; \mathcal{D}_{val}) - U(P_{\text{old}}, C; \mathcal{D}_{val})$
  if $\Delta U_P \geq \epsilon$ **then**
    continue ▷ Prompt evolution is still useful; do not edit control logic
  end if
▷ **Constrained control logic evolution**
  Analyze failures of (P, C) on $\mathcal{D}_{val}$
  Classify failures into structural categories, e.g., parsing errors, validation failures, retry failures, or inference-configuration issues
  Generate a constrained structural candidate C' within the safe-to-edit search space
  if $Cost(P, C') > B$ **then**
    Reject C'
    continue
  end if
▷ **Empirical validation validation**
  Evaluate old agent $A_{old} = (P, C)$ and candidate agent $A_{new} = (P, C')$ on the same validation subset $\mathcal{V} \subseteq \mathcal{D}_{val}$
  Compute $\Delta U_C \leftarrow U(A_{new}; \mathcal{V}) - U(A_{old}; \mathcal{V})$
  if $\Delta U_C > \delta$ **then**
    Accept the structural update: $C \leftarrow C'$
    Re-enter prompt evolution under the updated control logic
  else
    Reject C' and keep C unchanged
  end if
  Update $U_{best} \leftarrow U(P, C; \mathcal{D}_{val})$ if improved
end for
return $A^* = (P, C)$

- *Prompt evolution adds modest inference overhead.* Optimized prompts use 1.1–1.6× baseline tokens across the four benchmarks without additional model calls. On Qwen3.5-9B MMLU, PE improves accuracy by 1.4 percentage points; the subsequent structural update adds 5.7 points.
- *Structural evolution trades tokens for accuracy.* The evolved MMLU solver issues up to $3{+}1$ calls (3-way sample + conditional verification), while the IFEval solver issues up to $3{+}3{+}1$ calls (3-way sample + 3 self-checks + conditional repair). The MMLU multiplier is kept moderate ($\sim$3.6×) because the early-exit on consensus fires $\sim$80% of the time, avoiding the verification call. IFEval is more expensive ($\sim$5.1×) because the self-check scoring requires a separate LLM call per candidate, and the repair pass generates a full-length response. However, the accuracy gain on IFEval is substantially larger (+4.3 pp from +PE to +PACE on Qwen3.5-9B), reflect-

Table 10: Estimated per-query inference token breakdown for baseline and evolved solvers. *Input* counts prompt tokens (system + user + context); *Output* counts generated tokens. Multipliers are relative to the single-call baseline. Token counts are derived from code-level analysis of prompt templates and observed output lengths (marked with ∼); accuracy figures are from real evaluation runs on the respective test splits.

Benchmark	Solver variant	Input	Output	Total	Multiplier
MMLU	Baseline (single call)	∼350	∼120	∼470	1.0×
	+PE (optimized prompt)	∼420	∼150	∼570	1.2×
	+PACE (voting + verify)	∼1 260	∼450	∼1710	3.6×
IFEval	Baseline (single call)	∼280	∼350	∼630	1.0×
	+PE (optimized prompt)	∼340	∼380	∼720	1.1×
	+PACE (score + repair)	∼1700	∼1500	∼3200	5.1×
MGSM	Baseline (single call)	∼130	∼160	∼290	1.0×
	+PE (optimized prompt)	∼275	∼200	∼475	1.6×
	+PACE (translate + vote + verify)	∼420	∼1 100	∼1 520	5.2×
HotpotQA	Baseline (single call)	∼1 565	∼180	∼1 745	1.0×
	+PE (optimized prompt)	∼1 780	∼200	∼1 980	1.1×
	+PACE (answer refinement)	∼2 390	∼320	∼2 710	1.6×

Table 11: Estimated total token budget for the evolution process (PE and CE), broken down by phase. All figures are for Qwen3.5-9B with $K_{\max}$=20 steps, extrapolated from code-level analysis of a single evolution run.

Phase	MMLU	IFEval	MGSM	HpQA	Description
Prompt optimization	∼1.2M	∼1.8M	∼1.2M	∼2.4M	PE candidates × train/valid evals
Evaluation calls	∼0.4M	∼0.6M	∼0.5M	∼0.5M	Mini-batch (50 samples) + test-set evals
Controller reasoning	∼0.3M	∼0.3M	∼0.4M	∼0.6M	Conversation, tool calls, failure analysis
Structural exploration	∼0.2M	∼0.3M	∼0.4M	∼0.4M	validation comparisons, solver executions
Total evolution	**∼2.1M**	**∼3.0M**	**∼2.5M**	**∼3.9M**	One-time, amortized over future queries

ing the high value of structural intervention for constrained-generation tasks. MGSM exhibits the highest per-query multiplier (∼5.2×) due to its translate-then-solve pipeline plus N=5 self-consistency voting, but this cost is justified: majority voting over diverse reasoning paths catches stochastic arithmetic errors that no single prompt can eliminate, yielding +13.8 pp over Vanilla on Ministral-14B. In contrast, HotpotQA's evolved solver adds only a lightweight second extraction round with question-type-aware routing (∼1.6×), because the dominant failure mode—answer over-specification—is largely addressable through prompt refinement alone; the structural contribution is a yes/no detector that forces exact "yes"/"no" output for comparison questions, eliminating a failure mode worth ∼3 F1 points that prompt wording alone could not resolve.

Crucially, these multipliers are *discovered* by the agent, not prescribed. The Pareto-front optimization in the PE and the validation gate in the CE phase jointly penalize candidates that increase cost without proportional accuracy gain, steering the agent toward architectures with built-in cost control (early exits, conditional refinement).

Evolution-time token budget. Table 11 reports the estimated total tokens consumed during a single evolution run. The dominant cost is prompt optimization (∼60% of the budget), which evaluates multiple candidate configurations on train and validation splits across several iterations. The controller reasoning (agent conversation history, tool-call arguments, failure analysis summaries) accounts for only ∼15%, demonstrating that the framework's token-trimming mechanism (`_trim_messages_to_budget`) effectively controls context growth over long evolution trajectories.

The estimated evolution budget of 2.1–3.9M tokens is an upfront cost. Its per-query contribution decreases as deployment volume grows, while the evolved solver continues to incur the higher recurring inference cost in Table 10. This accounting does not establish a cost break-even against Vanilla or fine-tuning.

Table 12: Accuracy–cost Pareto trajectories for MMLU and IFEval using Qwen3.5-9B. Each row represents a solver variant: baseline, prompt-evolved (+PE), and fully evolved (+PACE). ΔAcc reports the incremental gain over the previous row within each benchmark. Efficiency is the incremental accuracy gain per 1,000 additional tokens. Accuracy values are taken from Table 1; token counts are estimates.

Benchmark	Variant	Tokens/query	Accuracy	ΔAcc (pp)	Eff. (pp/1k tok)
MMLU	Baseline	470	0.818	–	–
	+PE	570	0.832	+1.4	14.0
	+PACE	1710	0.889	+5.7	5.0
IFEval	Baseline	630	0.697	–	–
	+PE	720	0.718	+2.1	23.3
	+PACE	3200	0.761	+4.3	1.7
MGSM	Baseline	290	0.858	–	–
	+PE	475	0.872	+1.4	7.6
	+PACE	1 520	0.909	+5.1	4.1
HotpotQA	Baseline	1 745	0.754	–	–
	+PE	1 980	0.789	+3.5	14.9
	+PACE	2 710	0.803	+1.4	1.9

Why token efficiency matters for SLMs. The token analysis reveals a cost structure qualitatively different from the frontier-model setting. First, *output tokens dominate*: SLMs produce longer reasoning chains than frontier models to reach comparable accuracy, making output-token efficiency a first-order concern that the Pareto-front optimization directly addresses. Second, *batched sampling amortizes latency*: both evolved solvers use $n>1$ in a single API call; on vLLM, batched sampling with $n=3$ is only $\sim$1.3–1.5$\times$ slower than $n=1$ thanks to KV-cache sharing, so the cost sensitivity of the validation gate implicitly favors batched over sequential designs. Third, the *evolution cost is a fixed upfront investment* (2.1–3.9M tokens); once the solver is evolved it runs at the per-query cost in Table 10 indefinitely, an amortization property especially attractive for SLM deployments serving many queries on dedicated hardware. Finally, *credit assignment controls cost growth*: the ε-gated PE$\rightarrow$CE transition (Section 4.3) exhausts cheap prompt-level gains before incurring multi-call structural changes, avoiding the premature cost inflation visible in the $\varepsilon=1.0$ row of Table 4.

Serving latency and scope. Under the same vLLM/A100 configuration, measured relative MMLU latencies for Vanilla, +PE, +CE, and PACE are 1.00/1.05/1.25/1.30$\times$, respectively (Table 2). PACE uses approximately 1,710 tokens/query and 1.2 batched serving requests/query. These measurements characterize one serving configuration and do not establish universal latency, monetary-cost, or fine-tuning advantages.

A.7 Cross-Benchmark Solver Analysis: MMLU vs. IFEval

The structural evolution trajectories on MMLU and IFEval reveal that the controller agent independently discovers qualitatively different inference strategies tailored to each task's evaluation semantics, without any human-specified architectural prior. We analyse the two evolved solvers side by side to highlight how a frozen SLM, acting as its own architect, adapts control logic to the structure of the problem.

Shared meta-strategy. Despite targeting fundamentally different evaluation protocols, both evolved solvers converge on the same three-phase skeleton: *diverse generation* $\rightarrow$ *selection* $\rightarrow$ *conditional refinement*. This is not prescribed by the framework; the agent arrives at it independently on each benchmark. The convergence suggests that *generate–evaluate–refine* is a natural attractor in the space of inference-time strategies discoverable by SLMs, analogous to how self-consistency and chain-of-thought emerge as effective patterns in the prompting literature [Wang et al., 2023, Wei et al., 2022].

Task-adaptive specialization. Within the shared skeleton, the agent makes task-specific design choices that a human engineer would recognize as sensible:

- **Selection mechanism.** MMLU admits a trivial aggregation (majority vote over four labels), whereas IFEval requires evaluating each candidate against a heterogeneous constraint set. The agent replaces voting with an LLM-as-judge self-check that enumerates constraints and scores

Table 13: Structural comparison of evolved solvers on MMLU and IFEval. Both are discovered autonomously by the same controller agent (Qwen3.5-9B) via `action_adjust_logic`.

Dimension	**MMLU (classification)**	**IFEval (constrained generation)**
Answer space	Closed set $\{A, B, C, D\}$	Open-ended text subject to k verifiable constraints
Phase 1: Diversity	n=3 samples at $T \geq 0.5$; extract discrete labels	n=3 samples at $T \geq 0.3$; retain full response text
Selection criterion	Majority vote ($\geq 2/3$ consensus)	Constraint-satisfaction score $s \in [0, 1]$; pick $\arg\max_i s_i$
Early exit	Consensus $\Rightarrow$ return immediately	s=1.0 (all constraints pass) $\Rightarrow$ return immediately
Phase 2: Refinement	Verification tiebreak: present top answer + reasoning, ask model to confirm or correct at T=0	Targeted repair: feed best candidate back as assistant turn, ask model to rewrite satisfying all constraints at T=0
Acceptance gate	Accept verified answer if valid label	Accept repair only if $s_{\text{repair}} \geq s_{\text{best}}$
Output format	JSON with `reasoning`, `answer` keys	Raw text (`response_format="text"`) to avoid format-constraint conflicts
Avg. LLM calls	$\sim$1.3$\times$ (3-way batch + 15–20% verification)	$\sim$2.0$\times$ (3-way batch + scoring + conditional repair)

compliance— effectively inventing a lightweight verifier without access to the ground-truth checker.

- **Refinement strategy.** For MMLU, a wrong answer is best addressed by re-examining the reasoning chain (verification). For IFEval, a constraint violation is best addressed by rewriting the offending passage (repair). The agent discovers this distinction: it uses a *confirm-or-correct* prompt for MMLU and a *rewrite-to-satisfy* prompt for IFEval.
- **Output format.** The MMLU solver requests structured JSON to facilitate answer extraction, while the IFEval solver explicitly uses `response_format="text"` to avoid introducing formatting artefacts that would violate content constraints. This choice reflects awareness of the interaction between output format and evaluation criteria.
- **Temperature regime.** The MMLU solver uses $T \geq 0.5$ for diversity (reasoning paths are more varied at higher temperature), while the IFEval solver uses $T \geq 0.3$ (lower diversity suffices because constraint violations are structural, not stochastic).

Implications for SLM-driven self-evolution. The cross-benchmark comparison surfaces several properties that distinguish SLM self-evolution from conventional prompt engineering or human-designed inference pipelines:

1. **Emergent architectural transfer.** The generate–select–refine skeleton emerges independently on both tasks, suggesting that SLMs possess implicit knowledge of effective inference-time compute patterns. This is notable because the 9B model was never explicitly trained on meta-reasoning about solver design; the pattern is *discovered* through the interplay of failure analysis, credit assignment, and code generation.
2. **Constraint-aware adaptation without oracle access.** On IFEval, the agent invents a self-check scoring mechanism that approximates the ground-truth verifier (`verify_all_instructions`) without ever observing it. The self-check is imperfect—it relies on the same frozen model that generated the response—but it provides a sufficient signal to rank candidates and gate repairs. This demonstrates that SLMs can construct *task-specific evaluation proxies* as part of structural evolution, partially compensating for the absence of external reward models.
3. **Cost-aware design under implicit budgets.** Both solvers include early-exit conditions (consensus for MMLU, full-pass for IFEval) that avoid unnecessary refinement calls. The agent is not given an explicit cost objective, yet it discovers short-circuit logic that reduces average inference cost. This suggests that the validation gate mechanism, which rejects candidates that increase cost without proportional accuracy gain, implicitly teaches the agent to prefer efficient architectures.
4. **Robustness through conservative acceptance.** The IFEval solver accepts a repair only if it scores at least as well as the best candidate, preventing the repair pass from introducing new

Evolved MMLU solver (self-consistency + verification)

```
def solver(agent, task):
    cfg = normalize_prompt_config(agent.prompt_config)
    msg = [{"role": "user", "content": task}]

    # Phase 1: 3-way self-consistency (temp >= 0.5)
    resps = call_llm(msg, temp=max(cfg["T"], 0.5), n=3)
    votes, best = {}, {}
    for r in resps:
        a = extract_answer(r)           # coerce to A/B/C/D
        if a in {"A","B","C","D"}:
            votes[a] = votes.get(a, 0) + 1
            best.setdefault(a, r)

    top = max(votes, key=votes.get)
    if votes[top] >= 2:               # consensus
        return best[top]

    # Phase 2: verification tiebreak (temp = 0)
    v = call_llm(
        f"Previous answer: {top}
nReasoning: "
        f"{best[top]['reasoning'][:600]}
n"
        "Confirm or correct.", temp=0.0, n=1)
    return extract_answer(v[0]) or best[top]
```

Figure 4: Simplified evolved solver for MMLU. `call_llm` wraps `action_call_json_format_llm`; `extract_answer` wraps the coercion and regex extraction pipeline.

violations. Similarly, the MMLU solver falls back to the plurality answer when verification produces an invalid label. Both patterns reflect a *do-no-harm* principle that the agent learns from the empirical validation loop: edits that cannot be shown to help are rejected.

Unique strengths of SLM-powered evolution. Taken together, these observations highlight capabilities that are distinctive to the self-evolving SLM paradigm:

- **Zero-shot solver design.** The agent synthesises multi-phase inference strategies from scratch, without few-shot examples of solver code or access to a library of known techniques. A 9B model autonomously rediscovers self-consistency Wang et al. [2023] for MMLU and invents a generate–score–repair loop for IFEval—strategies that required separate research efforts when designed by humans.
- **Benchmark-specific structural creativity.** Rather than applying a single universal strategy, the agent tailors control flow, output format, temperature, and refinement logic to the evaluation protocol of each benchmark. This level of task-specific adaptation is difficult to achieve with prompt-only methods or fixed inference pipelines.
- **Self-improving without stronger teachers.** The entire evolution loop—prompt mutation, failure analysis, code generation, and empirical validation—is executed by the *same* frozen SLM that serves as the inference engine. No larger teacher model, human feedback, or gradient updates are required. The improvement comes purely from better *use* of the model's existing capabilities through evolved control logic.
- **Graceful degradation via empirical gating.** The credit-assignment and validation gate mechanisms ensure that structural exploration is both *triggered* and *validated* empirically. When the SLM proposes a flawed edit—which happens frequently at 9B scale—the gate rejects it without degrading the current solver. This makes the evolution process monotonically non-regressing in expectation, a property that is critical for deploying autonomous agents in practice.

Table 13 and Figures 4–5 together illustrate that the PACE framework enables a single frozen SLM to act as both the *subject* and the *architect* of its own inference pipeline, discovering task-appropriate strategies that would traditionally require separate human engineering effort for each benchmark.

A.8 τ-bench Evolution Analysis

We provide a detailed analysis of the PACE evolution trajectory on τ-bench Retail using Qwen3.5-9B, tracing both the failure taxonomy shifts and the accuracy progression across evolution phases.

Evolved IFEval solver (constraint scoring + targeted repair)

```
def solver(agent, task):
    cfg = normalize_prompt_config(agent.prompt_config)
    sys = build_system_prompt(cfg)
    msg = [{"role":"system","content":sys},
           {"role":"user","content":task}]

    # Phase 1: generate 3 candidates (temp >= 0.3)
    resps = call_llm(msg, temp=max(cfg["T"], 0.3),
                     n=3, fmt="text")
    candidates = [extract_text(r) for r in resps]

    # Phase 2: pick best via constraint self-check
    best, best_score = None, -1.0
    for c in candidates:
        s = self_check_constraints(agent, task, c)
        if s > best_score:
            best, best_score = c, s
    if best_score >= 1.0:          # all constraints pass
        return {"response": best}

    # Phase 3: targeted repair (temp = 0)
    repair_msg = msg + [
        {"role":"assistant","content":best},
        {"role":"user","content":
         "The response above violates some constraints."
         " Rewrite so EVERY constraint is satisfied."
         " Output ONLY the corrected response."}]
    fixed = call_llm(repair_msg, temp=0.0, n=1, fmt="text")

    # Accept repair only if it improves
    fix_s = self_check_constraints(
        agent, task, extract_text(fixed[0]))
    if fix_s >= best_score:
        return {"response": extract_text(fixed[0])}
    return {"response": best}
```

Figure 5: Simplified evolved solver for IFEval. `self_check_constraints` prompts the model to enumerate constraints and score compliance; `call_llm` wraps `action_call_llm` with `response_format="text"`. Compare with the MMLU solver in Figure 4.

A.8.1 Failure Taxonomy

Table 14 reports the failure category breakdown at each evolution checkpoint. The τ-bench evaluation framework classifies failures into four categories: *missing_actions* (the agent omitted a required tool call), *extra_actions* (the agent invoked an unnecessary or incorrect tool), *tool_errors* (the agent called a tool with invalid arguments, e.g., querying a non-existent user), and *early_stop* (the user simulator terminated the conversation prematurely due to agent confusion or unresponsiveness).

Table 14: Failure taxonomy across evolution phases for Qwen3.5-9B on τ-bench Retail (115 tasks, single-trial mid-loop evaluation). Categories are non-exclusive: a single failed task may exhibit multiple failure types (e.g., an extra tool call that also triggers a tool error and leads to early user termination).

Phase	**Pass**	**Fail**	**Missing**	**Extra**	**Tool Err**	**Early Stop**
Baseline (vanilla)	90	25	9	12	9	18
+ PE	93	22	9	10	11	21
+ PACE w/ structural edit (best)	96	19	7	11	4	16

Three patterns emerge from the taxonomy progression:

1. *Tool errors are halved by structural evolution.* Tool errors increase slightly under PE (9→11) but drop sharply to 4 after the structural edit, representing the largest relative reduction among all categories. This confirms that structural evolution addresses failure modes that prompt refinement cannot: the evolved control logic adds pre-call validation (e.g., verifying order status before executing irreversible actions) and enforces the single-call-per-order constraint that the SLM frequently violates under prompt-only guidance.

2. *Early stop remains the dominant failure mode throughout.* Early termination accounts for 70–80% of failures at every checkpoint. These failures arise when the stochastic user simulator loses patience with the agent's responses — a signal that is difficult to address through either prompt or structural changes alone, as it depends on the model's conversational fluency and the simulator's tolerance threshold.
3. *Missing actions decrease only after structural intervention.* Prompt evolution leaves missing_actions unchanged (9→9), while the structural edit reduces it to 7. The evolved solver's requirement to "collect ALL items into a single list before making the call" prevents the common failure pattern where the agent processes items sequentially and exhausts the one-call-per-order limit before completing all user requests.

A.8.2 Evolution Trajectory

Figure 6 visualizes the full evolution trajectory.

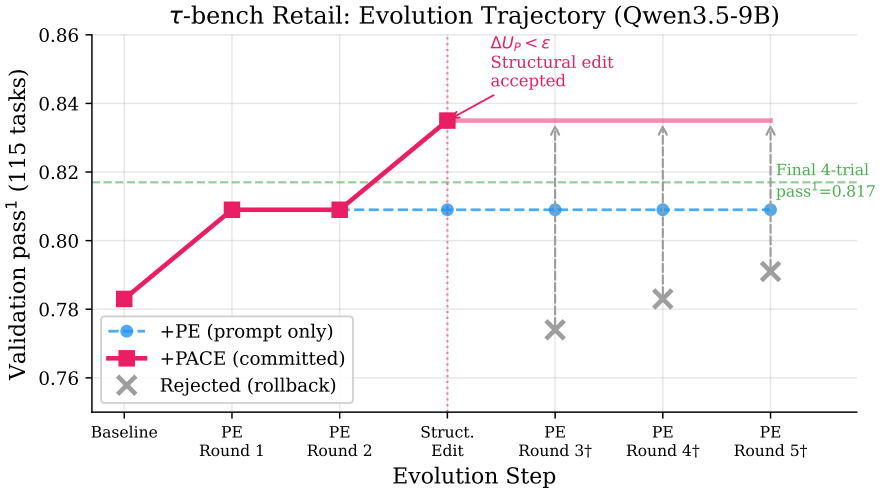

Figure 6: Evolution trajectory for Qwen3.5-9B on τ-bench Retail (115 tasks, single-trial validation during evolution). The +PE baseline (blue, dashed) saturates at 0.809 after two rounds. PACE (red, solid) matches PE during the prompt phase, then achieves a discrete jump to 0.835 when the structural edit is accepted at step 3. Subsequent PE rounds (gray crosses) regress and are rejected by the validation gate, preserving the post-structural peak. The green dashed line marks the final 4-trial pass^1=0.817.

The trajectory reveals several characteristics specific to multi-turn agent evolution:

1. *A single structural edit provides the largest discrete jump.* The accepted edit (step 3) improves validation accuracy from 0.809 to 0.835 (+2.6 pp), the largest single-step gain in the trajectory. The edit adds tool-call pre-validation and enforces the single-call-per-order constraint, directly addressing the `tool_errors` and `missing_actions` categories identified in the failure taxonomy.
2. *The validation gate prevents post-peak regression.* Steps 4–6 show that subsequent PE rounds after the structural edit consistently produce worse results (0.774, 0.783, 0.791), all rejected by the rollback mechanism. This demonstrates the monotonic non-regression property of PACE: once a good structural configuration is found, the gate prevents destabilizing changes.
3. *Multi-trial evaluation is lower than single-trial mid-loop.* The final 4-trial evaluation (pass^1=0.817) is lower than the best validation score (0.835). This gap reflects the inherent variance of multi-turn dialogue with a stochastic user simulator: a solver that passes 96/115 tasks in one trial may fail different subsets in subsequent trials. The pass^4 improvement over vanilla (0.661 vs. 0.626) confirms that PACE's structural edits improve robustness across trials, not just peak single-run performance.

A.9 Additional Baseline, Uncertainty, and Rejected-Edit Details

Baseline edit permissions. All adaptive methods share the frozen backbone, controlled data splits, task metrics, and 20-step evolution budget. These comparisons match evolution steps rather than total tokens or wall-clock time. ACE modifies textual context while retaining a fixed execution harness; Gödel Agent permits broader executable self-modification. The closest edit-space-matched comparator is +CE, which uses PACE's admissible control-logic space and validation gate but enables structural search immediately and omits prompt evolution.

Table 15: Baseline adaptation permissions. CL denotes control logic. "–" denotes a condition without CL updates.

Method	Prompt edits	CL edits	PACE CL space	CL validation	Steps
Vanilla	No	No	–	–	0
+PE	Yes	No	–	–	20
+CE	No	Yes	Yes	Yes	20
ACE	Yes	No	–	–	20
Gödel Agent	Yes	Yes	Broader	No	20
PACE	Yes	Yes	Yes	Yes	20

Run-level uncertainty. Table 16 reports Welch 95% confidence intervals for differences between means across ten independent evolution runs, computed from the reported run-level means and standard deviations. They quantify evolution-run variability under the fixed evaluation protocol, rather than uncertainty over all possible benchmark samples or deployment distributions. All six intervals exclude zero. Without matched seed-level outcomes, these are unpaired comparisons.

Table 16: Qwen3.5-9B gains and Welch 95% confidence intervals, in percentage points.

Benchmark	Comparison	Gain (pp)	95% CI (pp)
MMLU	PACE − Vanilla	7.1	[6.5, 7.7]
MMLU	PACE − PE	5.7	[5.0, 6.4]
MMLU	PACE − CE	5.4	[4.0, 6.8]
IFEval	PACE − Vanilla	6.4	[5.4, 7.4]
IFEval	PACE − PE	4.3	[3.2, 5.4]
IFEval	PACE − CE	3.6	[1.4, 5.8]

Representative harmful proposals. A strict JSON-only MMLU pipeline with regex- and position-based answer extractors discarded valid free-form answers and forced incorrect option letters, reducing validation accuracy from .83 to .00. An IFEval post-processor tried to enforce keywords, length, and Markdown requirements by prepending corrective boilerplate; meta-text such as "Missing required keyword" introduced new violations and reduced validation accuracy from .58 to .42. Both executable proposals were rejected by validation. On MGSM/HotpotQA, a multi-stage reasoning and verification proposal with 7–10 fallback extractors contained malformed f-strings and regex literals and was rejected as non-executable. These examples show why executable syntax or model self-assessment alone is insufficient to accept an edit.

Scope of mechanism attribution. PACE evolves interacting task-specific logic: one proposal can jointly alter parsing, decoding, verification, and retry behavior. Consequently, accepted-edit counts characterize the search process but do not isolate additive performance contributions. A causal decomposition would require controlled interventions on each final solver, including interactions among components.

Empirical stability and scope. On IFEval, immediate structural search ($\varepsilon = 1.0$) reaches .725, compared with .761 for the saturation-gated default; disabling the validation gate reduces performance to .723 (Table 4). In the Retail trajectory (Appendix A.8), an accepted update improves single-trial validation from .809 to .835, while subsequent regressing prompt updates (.774/.783/.791) are rolled back. These are empirical observations under the reported protocol, not formal stability or safety guarantees. The larger-model scaling check uses HotpotQA F1 as reported in the additional

evaluation; its absolute scores should not be equated with the exact-match objective of the controlled SLM suite.

Rejected proposal causes and accepted mechanisms. The proposal audit in Table 5 assigns each rejection a primary cause (Table 17). Accepted proposals comprise parsing/normalization (2; 10.5%), retry/repair (6; 31.6%), voting/self-consistency (8; 42.1%), verification (2; 10.5%), and routing/fallback (1; 5.3%); no validation-only proposal was accepted. The high acceptance rates for retry/repair and routing/fallback are based on only six and one proposals, respectively.

Table 17: Primary causes among the 68 rejected control-logic proposals.

Primary cause	Count	Fraction of rejected proposals
Non-executable / runtime failure	15	22.1%
Interface or output-contract violation	0	0.0%
Validation regression	20	29.4%
Improvement below acceptance margin	15	22.1%
Resource-budget violation	18	26.5%
Total	68	100.0%

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 claim that PACE improves SLM performance through two-timescale prompt and control-logic evolution. These claims are directly supported by the experimental results in Table 1 and the ablation study in Section 4.3. Limitations regarding model scale and benchmark coverage are discussed.

2. **Limitations**

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

   Answer: [Yes]

   Justification: Section 5 discusses benchmark coverage, absent multimodal evaluation, larger-model non-improvement, evolution and serving costs, repeated-validation selection bias, and the absence of formal convergence or general safety guarantees. The estimated upfront evolution cost is 2.1–3.9M tokens.

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 is primarily empirical. It does not present formal theorems or proofs. The credit-assignment threshold ε and A/B gate δ are design choices validated experimentally rather than derived theoretically.

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 specifies the SLM backbones (Qwen3-4B, Qwen3.5-9B, Ministral-14B), the serving infrastructure (vLLM), all benchmarks (MMLU, MGSM, HotpotQA, IFEval), data splits, hyperparameters (ε, δ, temperature, n), and the evolution procedure in sufficient detail to reproduce the main 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: The PACE implementation is available at `https://github.com/lingchen0331/PACE-self-evolving-agent`, also linked in the abstract. The paper specifies the algorithm (Algorithm 1), hyperparameters, data splits (Table 7), and evolved solver listings (Figures 4–5); benchmarks and models are publicly available.

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: Section A.1 details the data splits, temperature settings, sampling parameters, and the Pareto-front optimization procedure. The ablation in Section 4.3 reports all swept hyperparameter values. Additional details appear in the appendix.

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 standard deviation is listed in Table 8.

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 that all SLMs are served locally via vLLM, describes the evolution budget (2–3M tokens per run), and provides per-query token breakdowns in Table 10. GPU type and wall-clock times are detailed in the appendix.

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 uses only publicly available benchmarks and open-weight language models. No private user data is collected or used. The work conforms with the NeurIPS Code of Ethics.

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 the positive impact of making capable agentic systems accessible via smaller, locally deployable models, reducing reliance on costly frontier APIs. It also acknowledges the risk that self-evolving agents could produce unexpected or undesirable behaviors if the gating mechanisms fail, and discusses mitigation through the A/B gate and human-inspectable evolution logs.

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 pre-trained models or scraped datasets. PACE is an optimization framework applied to existing open-weight models, and does not introduce new artifacts with high misuse risk.

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: All benchmarks (MMLU, MGSM, HotpotQA, IFEval) and models (Qwen3-4B, Qwen3.5-9B, Ministral-14B) are cited with their original publications. License information for the models and datasets is provided in the appendix.

13. **New assets**

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

    Answer: [N/A]

    Justification: The paper does not release new datasets or model checkpoints at submission time. The PACE framework code will be documented and released upon acceptance.

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: This work does not involve crowdsourcing or research with human subjects.

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: This work does not involve human subjects research and therefore does not require IRB approval.

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?

    Answer: [Yes]

    Justification: LLMs are central to the PACE framework: an SLM serves as both the inference backbone and the outer-loop optimizer that drives prompt and structural evolution. The paper fully describes which models are used (Qwen3-4B, Qwen3.5-9B, Ministral-14B), how they are deployed (vLLM), and their role in each component of the system.