TYLER: Typed Latent Reasoning for Language Models —— When to Think, What to Compute, and How Much to Allocate

Hanyu Lin[1*] **Min Cai**[2*] **Jiawei Wen**[1*] **Haodi Zhang**[1]
[1]Shenzhen University [2]University of Alberta

Abstract

Chain-of-thought (CoT) prompting improves reasoning in large language models (LLMs) by externalizing intermediate computation as discrete text tokens, but this textual interface also introduces redundancy and inference overhead. Latent reasoning offers a promising alternative by carrying part of the computation in continuous representations. However, existing methods typically predefine when latent computation is invoked and how it is allocated during decoding, leaving a key problem unresolved: when to invoke latent computation, what type of computation to perform, and how much budget to allocate. We propose **Ty**ped **L**at**e**nt **R**easoning (**TYLER**), a typed and budget-aware framework for latent reasoning during autoregressive decoding. **TYLER** learns a policy that, at each decoding step, chooses between emitting a text token and switching to a latent computation module specialized for a particular reasoning function. Once invoked, an operator maps the current reasoning state into latent tokens that support global planning, local state updates, or reusable procedural abstraction. Across extensive experiments on three backbone LLMs, **TYLER** improves accuracy by up to 14.49 points over CoT and by up to 4.30 points over the strongest competing baseline. It further generalizes across diverse reasoning domains and achieves the best final-stage performance with the lowest forgetting.

1 Introduction

Large Language models (LLMs) typically solve complex reasoning tasks by generating explicit chain-of-thought (CoT) token sequences (Wei et al., 2022; Wang et al., 2022; Jaech et al., 2024; Guo et al., 2025). Although CoT improves reasoning performance, it requires intermediate computation to be externalized as visible text. This requirement increases redundancy generation and inference cost.

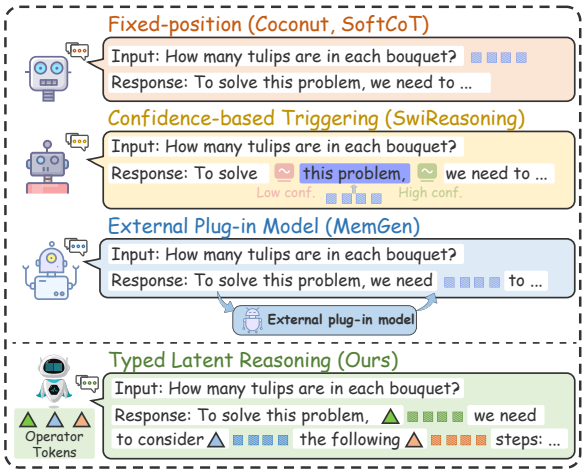

Figure 1: Comparison of latent reasoning paradigms. Existing methods rely on fixed-position latent tokens, confidence-triggered latent computation, or external trigger models. **TYLER** instead allows the LLM to interleave between visible decoding and typed latent-operator invocation during generation.

It also imposes a rigid interface: intermediate reasoning must be written out before the final answer is produced, rather than invoked internally on demand during autoregressive decoding.

Latent reasoning provides a promising way to relax this interface by keeping part of the intermediate computation in continuous representations, instead of unfolding the full reasoning process into explicit tokens (Zhu et al., 2025; Hao et al., 2022; Xu et al., 2025a; Zhang et al., 2026). From this perspective, latent reasoning should not be viewed merely as a compressed form of CoT. Rather, it can serve as a computation mechanism that allows the model to choose between visible decoding and silent latent computation during generation.

Despite encouraging progress, two limitations remain. First, prior work mainly focuses on *how* to construct latent tokens, such as from hidden states (Hao et al., 2022; Liu et al., 2026), mixtures over token-embedding distributions (Zhang et al.,

[*]Equal contribution.

2026; Shi et al., 2025), or auxiliary soft-thought generators (Xu et al., 2025a,b; Zhang et al., 2025). This construction-centered view often treats latent tokens as general-purpose carriers of intermediate computation. However, explicit reasoning chains are functionally heterogeneous: different steps may orient the problem, update the evolving reasoning state, or reuse procedural solution patterns. A latent reasoning mechanism should preserve these functional distinctions rather than collapse them into a single undifferentiated representation.

Second, existing methods remain limited in how they adaptively coordinate explicit reasoning and latent computation during generation. Fixed-position methods (Hao et al., 2022; Xu et al., 2025a) decouple latent computation from the evolving reasoning state, while adaptive methods mainly decide whether or where to enter latent space based on uncertainty signals (Shi et al., 2025; Liu et al., 2026) or an external trigger model (Zhang et al., 2025). They do not allow the LLM itself to decide, during autoregressive decoding, which type of latent computation to perform or how much computation budget to allocate. This gap motivates the following research question:

> *Can an LLM learn to interleave visible decoding with silent latent computation, while deciding when to think, what type of computation to perform, and how much budget to spend?*

To answer this question, we propose **TYLER**, a typed latent reasoning framework that learns when to invoke latent computation, which type of computation to perform, and how much budget to allocate. During generation, the LLM chooses between visible-token decoding and three typed latent operators: O_g for global orientation, O_s for state update, and O_p for procedural abstraction. Each operator maps the current reasoning state to a sequence of operator-conditioned latent tokens, which then guide subsequent visible decoding. Because these operators are defined by reusable computational roles rather than task-specific output formats, **TYLER** supports latent computation that can transfer across reasoning domains.

The training procedure consists of two stages. Stage 1 optimizes the latent operators and the Latent Synthesis Model while keeping the LLM frozen, aligning each operator with its intended reasoning function. Stage 2 optimizes an operator-invocation policy with Group Relative Policy Optimization (GRPO), updating the policy parameters while keeping the latent synthesis interface learned in Stage 1 fixed. An operator-anchored auxiliary objective further stabilizes policy learning at sparse invocation positions.

Empirically, **TYLER** achieves consistent improvements across multiple backbones and benchmarks, with gains that transfer beyond the training domain. Further analysis shows that the learned policy invokes different operators across task difficulties and reasoning stages, suggesting that these operators acquire differentiated functional roles. In a sequential adaptation setting spanning code, science, math, and theorem reasoning, **TYLER** obtains higher final performance with lower forgetting, indicating stronger continual adaptation.

Our contributions are summarized as follows:

- We formulate latent reasoning as an online, typed, and budgeted computation problem, where an LLM learns when to invoke latent computation, which operation to perform, and how much budget to allocate.
- We introduce **TYLER**, a typed latent reasoning framework that interleaves visible decoding with latent operators for global orientation, state update, and procedural abstraction.
- We conduct extensive experiments across multiple backbones and reasoning benchmarks, showing that **TYLER** improves accuracy over strong baselines, generalizes beyond the training domain, and exhibits stronger continual adaptation under sequential domain shifts.

2 Related Work

Explicit reasoning. CoT improves the reasoning ability of LLMs by making intermediate computation explicit as text (Wei et al., 2022). Subsequent work extends this paradigm through sampled reasoning paths (Wang et al., 2022), external actions and tool use (Yao et al., 2022; Schick et al., 2023), and search-based method (Yao et al., 2023). Reinforcement learning has also become central to scaling explicit reasoning behavior, ranging from PPO-based RLHF (Schulman et al., 2017; Ouyang et al., 2022) to recent methods based on GRPO-style optimization (Shao et al., 2024; Yu et al., 2026). These methods strengthen reasoning mainly by increase the computational load. In contrast, **TYLER** preserves the autoregressive decoding interface while allowing the model to adaptively choose between

emitting visible tokens and performing silent latent computation.

Latent reasoning. Latent reasoning replaces part of the visible tokens with continuous representations (Zhu et al., 2025; Chen et al., 2025). Existing methods mainly study how latent tokens are constructed, including recycled hidden states (Hao et al., 2022; Shen et al., 2025), soft thought tokens (Xu et al., 2025a; Wei et al., 2025), and embedding-space mixtures (Zhang et al., 2026). Other work studies when to activate latent computation through confidence signals (Shi et al., 2025; Liu et al., 2026), enternal trigger model (Zhang et al., 2025), or pause tokens (Goyal et al., 2024; Pfau et al., 2024). These methods demonstrate the value of latent reasoning, but they usually treat latent computation as a uniform form. They provide limited study for deciding what functional role latent computation should serve or how much computation budget should be allocated. In contrast, **TYLER** formulates latent reasoning as an online, typed, and budgeted computation problem, where latent operators serve different computational roles and are invoked under an explicit budget.

3 Methodology

3.1 Preliminaries

We consider a decoder-only autoregressive language model parameterized by θ, which defines a next-token distribution $p_\theta(\cdot \mid x_t)$ over a discrete vocabulary V given an input embedding sequence x_t. Let d denote the embedding dimension, and let $x_t \in \mathbb{R}^{T_t \times d}$ be the decoding context at step t, where T_t is the current sequence length. In standard autoregressive decoding, the model samples a visible token

$$a_t \sim p_\theta(\cdot \mid x_t), \tag{1}$$

and appends its embedding to form the next-step input:

$$x_{t+1} = [\, x_t;\, \mathrm{emb}(a_t)\,]. \tag{2}$$

This scheme tightly couples intermediate computation with discrete token generation: each reasoning state must be externalized through a vocabulary-indexed embedding before it can influence subsequent decoding. Latent reasoning addresses this constraint by allowing continuous vectors to be inserted into the decoding context. Given the current embedding sequence x_t, a set of latent tokens $z_t \in \mathbb{R}^{N_t \times d}$ can be appended as

$$x_{t+1} = [\, x_t;\, z_t\,]. \tag{3}$$

These latent tokens are consumed by the autoregressive model as internal computational states while remaining invisible to the user.

3.2 Overview

As shown in Figure 2, method extends autoregressive decoding with adaptive latent computation. The framework consists of two main components. First, the operator invocation policy decides whether the model should emit a visible token or invoke a typed operator (Section 3.3). Second, the latent synthesis module maps the activated operator and the current context to a sequence of latent tokens (Section 3.4).

3.3 Operator Invocation

We first define the latent operators and describe how **TYLER** invokes them during autoregressive decoding. In explicit CoT reasoning, intermediate tokens often play different functional roles: some provide global orientation, some update the local reasoning state, and others express reusable procedural patterns. This suggests that latent computation should not be treated as a single undifferentiated operation.

Motivated by this observation, **TYLER** introduces three typed latent operators. Each operator O_k has learnable parameters ω_k and maps the current decoding context to a sequence of latent tokens that supports subsequent generation:

$$O_k(\cdot;\omega_k) : x_t \mapsto z_t^k \in \mathbb{R}^{N_k \times d}. \tag{4}$$

The operator type provides an inductive bias about the intended computational role, while its behavior is learned from CoT data.

We instantiate three operators:

$$\mathcal{O} = \{O_g, O_s, O_p\}, \tag{5}$$

where O_g supports global orientation and planning, O_s supports local state updates, and O_p captures reusable procedural abstractions. These operators are encouraged by heuristic position supervision in Stage 1 (Section 4.1) and regulated by a budget-aware routing policy in Stage 2 (Section 4).

To enable operator invocation during decoding, we introduce one special token for each operator. Let $\mathcal{U} = \{u_g, u_s, u_p\}$ denote the set of operator tokens, where u_k is associated with O_k. The decoding action space is extended to $V' = V \cup \mathcal{U}$

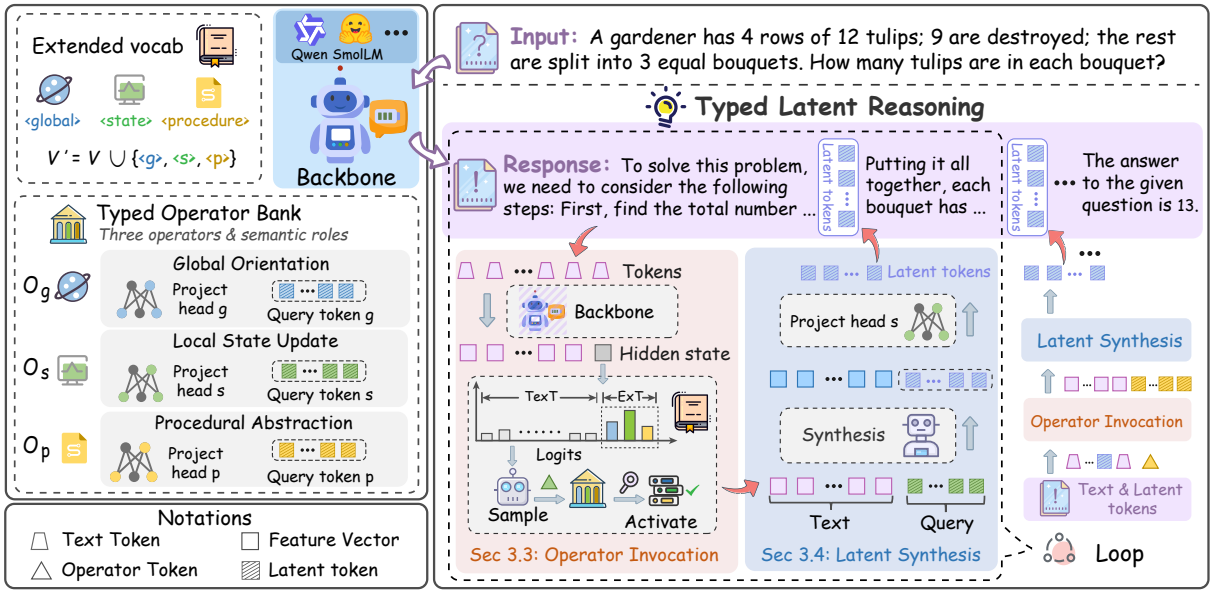

Figure 2: Overview of **TYLER**. During decoding, the backbone interleaves between visible text tokens and typed operators. When invoked, each operator maps the current context to an operator-conditioned sequence of latent tokens, which is inserted back into the context to guide subsequent decoding.

by adding three trainable operator-token rows $\psi \in \mathbb{R}^{3\times d}$ to the LM head. The next-token distribution is denoted as $p_{\theta,\psi}(\cdot \mid x_t)$, and the model samples

$$a_t \sim p_{\theta,\psi}(\cdot \mid x_t), \qquad a_t \in V'. \tag{6}$$

If a_t is a visible token, the model appends its embedding and emits it to the user. If $a_t = u_k$ is an operator token, the corresponding operator synthesizes latent tokens $z_t^k = O_k(x_t; \eta_k)$, which are appended to the context x_t:

$$x_{t+1} = \begin{cases} [\,x_t;\, \mathrm{emb}(a_t)\,], & a_t \in V, \\ [\,x_t;\, z_t^k\,], & a_t = u_k \in \mathcal{U}. \end{cases} \tag{7}$$

The same policy is then applied to the updated context x_{t+1}. Thus, visible-token emission and latent-operator invocation compete within a single next-action distribution, without requiring an external controller.

3.4 Latent Synthesis

Once the LLM samples an operator O_k, **TYLER** must synthesize latent tokens that are compatible with the LLM input-embedding space. Rather than directly reusing hidden states (Hao et al., 2022) or mixing vocabulary-space embeddings (Zhang et al., 2026), we follow MemGen (Zhang et al., 2025) and add trainable LoRA layers (Hu et al., 2022) to the backbone while keeping the original parameters θ frozen. Thus, latent tokens are synthesized through parameter-efficient adaptation without updating the backbone directly.

When O_k is activated at step t, it uses the current context x_t to produce N_k latent tokens. All operators share the LoRA-augmented synthesizer $\mathrm{Synth}_{\theta,\phi}$, but specialize through operator-specific query tokens $Q_k \in \mathbb{R}^{N_k\times d}$ and projection heads $\mathrm{Proj}_k : \mathbb{R}^d \to \mathbb{R}^d$.

The activated operator appends its query tokens to the current context and performs a single forward pass through the LoRA-augmented LLM. Because the synthesizer is decoder-only and the query tokens are placed after the context, the query positions can attend to the preceding context:

$$h_t^k = \mathrm{Synth}_{\theta,\phi}([\,x_t;\, Q_k\,])_{\text{query}} \in \mathbb{R}^{N_k\times d}. \tag{8}$$

The operator-specific projection head then maps the query-position hidden states into the LLM input-embedding space:

$$z_t^k = \mathrm{Proj}_k(h_t^k) \in \mathbb{R}^{N_k\times d}. \tag{9}$$

Although all operators share Synth_ϕ and read the same context x_t, they specialize through distinct query tokens Q_k, distinct projection heads Proj_k, and context-aware supervision over invocation positions introduced in Section 4.1.

4 Training

TYLER is trained in two stages. Stage 1 learns to synthesize operator-conditioned latent tokens while

keeping the backbone LLM fixed. Stage 2 freezes the latent synthesis-specific parameters learned in Stage 1 and trains the LLM to invoke typed operators adaptively under a computation budget. Complete training details are provided in Appendix F.

4.1 Latent Synthesis Optimization

Stage 1 optimizes the LoRA parameters ϕ in the latent synthesizer $\mathrm{Synth}_{\theta,\phi}$ and the operator-specific components $\{Q_k, \mathrm{Proj}_k\}_{O_k \in \mathcal{O}}$, while keeping the backbone LLM parameters θ frozen. We denote the trainable Stage 1 parameters by $\Phi = \{\phi, \{Q_k, \mathrm{Proj}_k\}_{O_k \in \mathcal{O}}\}$. For each supervised reasoning trace $(q, y_{1:T})$, we construct a set of candidate boundary–operator pairs $\mathcal{B}(q, y)$ using structure-aware heuristics. Specifically, O_g is assigned to answer-onset positions, O_s to reasoning-step boundaries, and O_p to structural boundaries such as formulas, code blocks, and structured fields. Details are provided in Appendix F.

Given a pair $(b, k) \sim \mathcal{B}(q, y)$, the selected operator synthesizes latent tokens $z_b^k = O_k(x_b)$ from the prefix $x_b = (q, y_{<b})$. We insert z_b^k into the teacher-forced trajectory, perform a single causal forward pass, and compute the next-token cross-entropy only at visible target-token positions:

$$\mathcal{L}_{\text{stage1}} = -\mathbb{E}_{\mathcal{D},\mathcal{B}} \sum_{t=1}^{T} \log p_{\theta,\Phi}\left(y_t \mid \tilde{c}_{b,t}^k\right). \quad (10)$$

Here, $\mathbb{E}_{\mathcal{D},\mathcal{B}}$ denotes the expectation over $(q, y) \sim \mathcal{D}$ and $(b, k) \sim \mathcal{B}(q, y)$. $\tilde{c}_{b,t}^k$ denotes the latent-augmented context before predicting the visible token y_t. For $t < b$, this context reduces to the visible prefix $(q, y_{<t})$; for $t \geq b$, it additionally contains the inserted latent tokens, i.e., $(q, y_{<b}, z_b^k, y_{b:t-1})$. The inserted latent tokens are used only as context and are not prediction targets. Gradients are applied only to Φ, while the backbone parameters θ remain frozen. Consequently, Stage 1 trains each operator through visible next-token supervision while preserving the backbone reasoner.

4.2 Operator Invocation Optimization

Stage 2 learns a budget-aware operator-invocation policy that decides whether to emit a visible token or invoke latent computation, which typed operator to select, and how much budget to spend. We freeze the Stage 1 synthesis parameters $\Phi = \{\phi, \{Q_k, \mathrm{Proj}_k\}_{O_k \in \mathcal{O}}\}$, including the latent synthesizer LoRA parameters and all operator-specific components. The backbone base weights θ are also kept frozen. The decoding policy is adapted through a separate set of LoRA parameters η, together with the operator-token head rows ψ. We denote the resulting policy distribution by $p_{\theta,\eta,\psi}$ and optimize $\{\eta, \psi\}$ with GRPO (Shao et al., 2024). For each prompt q, the policy samples a group of trajectories. Each trajectory is assigned a reward that combines task performance with a success-gated budget penalty P_{bud}:

$$\begin{aligned} P_{\text{bud}}(\tau) &= s(\tau)\big(n_{\text{op}}(\tau) - B_O\big)_+, \\ R(\tau) &= R_{\text{task}}(\tau) - \lambda P_{\text{bud}}(\tau). \end{aligned} \quad (11)$$

Here, $s(\tau)$ indicates whether the trajectory successfully solves the task, $n_{\text{op}}(\tau)$ is the number of operator invocations, and B_O is the invocation budget. The penalty is applied only to successful over-budget trajectories, encouraging the policy to use latent computation efficiently without discouraging exploration on failed trajectories.

Because operator invocations occupy only a small fraction of each generated trajectory, sequence-level rewards provide weak credit-assignment signals for operator selection. We therefore introduce an operator-anchored auxiliary objective that applies the GRPO surrogate at operator-invocation positions. The Stage 2 objective is

$$\mathcal{L}_{\text{stage2}} = \mathcal{L}_{\text{GRPO}}(R) + \alpha \mathcal{L}_{\text{anch}}, \quad (12)$$

where $\mathcal{L}_{\text{anch}}$ stabilizes optimization at sparse invocation positions, and α controls the strength of this auxiliary objective.

5 Experiments

We organize the evaluation around five questions: whether **TYLER** (i) generalizes across domains, (ii) retains continual-learning capabilities during sequential task adaptation, (iii) benefits from each proposed component, (iv) invokes operators in a task-adaptive manner under a computation budget, and (v) induces distinguishable functional roles among the three typed operators.

5.1 Experimental Setup

Backbones and training. We instantiate **TYLER** on Qwen2.5-1.5B-Instruct (Yang et al., 2024), SmolLM3-3B (Bakouch et al., 2025), and Qwen3-4B (Yang et al., 2025), covering three model families in the 1.5B–4B range. Stage 1 and Stage 2 are trained on 15K problem–solution traces sampled from OpenR1-Math (Hugging Face, 2025); the same data and prompt templates are used for all training-based baselines.

Backbone	Method	GSM8K	MATH-500	GPQA-Diamond	TheoremQA	Average
SmolLM3-3B	CoT	58.91	68.00	19.70	19.54	41.54 (↑0.00)
	SFT	83.02	66.60	21.72	25.30	49.16 (↑7.62)
	GRPO	72.71	<u>74.20</u>	28.28	26.51	50.43 (↑8.89)
	SoftCoT	83.85	69.20	24.24	25.03	50.58 (↑9.04)
	Soft-Thinking	76.65	68.40	21.72	21.15	46.98 (↑5.44)
	MemGen	<u>84.15</u>	70.20	25.25	27.30	51.73 (↑10.19)
	SwiReasoning	74.98	68.60	20.70	19.40	45.92 (↑4.38)
	TYLER Stage 1	83.24	71.20	<u>30.50</u>	<u>28.76</u>	<u>53.43</u> (↑11.89)
	TYLER Stage 2	**85.21**	**76.40**	**33.50**	**29.00**	**56.03** (↑14.49)
Qwen3-4B	CoT	90.52	82.60	16.16	36.95	56.56 (↑0.00)
	SFT	86.96	72.80	21.72	35.61	54.27 (↓2.29)
	GRPO	90.14	81.20	39.90	<u>39.36</u>	62.65 (↑6.09)
	SoftCoT	88.32	72.60	38.38	34.94	58.56 (↑2.00)
	Soft-Thinking	91.51	81.40	15.15	36.41	56.12 (↓0.44)
	MemGen	89.65	78.20	<u>40.40</u>	36.87	61.28 (↑4.72)
	SwiReasoning	90.21	82.00	17.17	33.50	55.72 (↓0.84)
	TYLER Stage 1	<u>91.66</u>	<u>83.00</u>	39.90	38.95	<u>63.38</u> (↑6.82)
	TYLER Stage 2	**92.87**	**85.80**	**43.43**	**41.02**	**65.78** (↑9.22)

Table 1: Results on **SmolLM3-3B** and **Qwen3-4B**. All values are Pass@1 accuracy (%). We highlight the **best** and <u>second-best</u> results. Improvements across both backbones indicate that TYLER generalizes well across multiple task domains.

Benchmarks. We evaluate on four benchmarks covering three reasoning regimes: mathematical reasoning with GSM8K (Cobbe et al., 2021) and MATH-500 (Hendrycks et al., 2021); scientific reasoning with GPQA-Diamond (Rein et al., 2023); and theorem-oriented reasoning with TheoremQA (Chen et al., 2023).

Baselines. We compare TYLER against representative methods in two regimes. (i) Explicit reasoning: CoT (Wei et al., 2022), SFT, and GRPO (Shao et al., 2024). (ii) Latent reasoning: SoftCoT (Xu et al., 2025a), Soft-Thinking (Zhang et al., 2026), MemGen (Zhang et al., 2025), and SwiReasoning (Shi et al., 2025).

Metrics and implementation. We report Pass@1 accuracy under greedy decoding, and compute the average as an unweighted macro-average across the four benchmarks. Unless otherwise specified, Stage 1 uses latent-token lengths $(N_g, N_s, N_p) = (8, 4, 4)$, and Stage 2 uses a latent-call budget $B_O = 5$. Implementation details are provided in Appendix E.

5.2 Main Results

Consistent Gains, Cross-Domain Generalization, and Robustness. TYLER achieves the best average performance on both backbones. On SmolLM3-3B, Stage 2 reaches an average accuracy of 56.03, outperforming CoT by 14.49 points and the strongest baseline, MemGen, by 4.30 points. On Qwen3-4B, Stage 2 reaches 65.78, improving over CoT by 9.22 points and over the strongest prior baseline, GRPO, by 3.13 points. Although TYLER is trained only on a subset of OpenR1-Math, its gains extend beyond mathematical reasoning to scientific and theorem-oriented benchmarks: Stage 2 improves GPQA-Diamond over CoT by 13.80 and 27.27 points on SmolLM3-3B and Qwen3-4B, respectively, and improves TheoremQA by 9.46 and 4.07 points. These results suggest that the learned latent computation transfers beyond the training domain. Existing baselines are more sensitive to the choice of backbone: SFT improves SmolLM3-3B but slightly degrades Qwen3-4B, while GRPO is stronger on Qwen3-4B than on SmolLM3-3B. Latent-reasoning baselines show similar variance, with MemGen being strongest on SmolLM3-3B but not on Qwen3-4B. In contrast, TYLER consistently achieves the best average performance and maintains gains on out-of-domain benchmarks, indicating stronger robustness across backbone choices and evaluation domains.

Continual Learning. Figure 3 evaluates TYLER under sequential adaptation across code, science, math, and theorem-oriented tasks. After the final adaptation stage, TYLER achieves the highest macro-average accuracy, reaching 39.8% and outperforming the strongest baseline by about 3.5

	Variant	Routing	Typed	Budget	$\mathcal{L}_{\text{anch}}$	GSM8K	MATH	GPQA	Theorem	Average
Per-stage	(a) CoT	–	–	–	–	90.52	82.60	16.16	36.95	56.56 (↑0.00)
	(b) +Stage 1	×	✓	–	–	91.66	83.00	39.90	38.95	63.38 (↑6.82)
	(c) +Stage 2 (task-only)	✓	✓	×	×	**93.25**	83.60	38.38	40.05	63.82 (↑7.26)
	(d) +Stage 2 (+budget)	✓	✓	✓	×	91.57	83.80	38.89	40.42	63.67 (↑7.11)
	(e) **TYLER**	✓	✓	✓	✓	92.87	**85.80**	**43.43**	**41.02**	**65.78** (↑9.22)
Typed?	(f) Shared operators	✓	×	✓	✓	91.10	83.20	37.88	39.62	62.95 (↑6.39)
	(g) Swap $O_s \leftrightarrow O_g$	✓	swap	✓	✓	92.49	80.40	35.35	33.50	60.44 (↑3.88)
Routing?	(h) Rand@25%	random	✓	≈	–	90.71	82.80	34.85	38.81	61.79 (↑5.23)
	(i) Rand@50%	random	✓	≈	–	91.01	83.00	35.86	39.22	62.27 (↑5.71)
	(j) Rand@75%	random	✓	≈	–	91.15	82.60	36.36	38.65	62.19 (↑5.63)
	(k) Rand@100%	random	✓	–	–	91.66	83.00	39.90	38.95	63.38 (↑6.82)

Table 2: Unified ablation on Qwen3-4B. All values are Pass@1 accuracy (%);

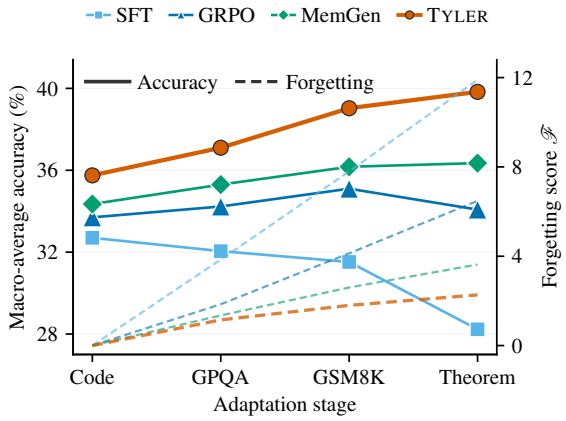

Figure 3: Continual-learning results on Qwen2.5-1.5B-Instruct. *(a)* Macro-average after each adaptation stage. *(b)* Forgetting score $\mathcal{F}$; lower is better.

points. It also obtains the lowest forgetting score, with $\mathcal{F} = 2.3$, reducing forgetting by about 1.4 points compared with MemGen. These results indicate that **TYLER** not only adapts more effectively to new reasoning domains, but also better preserves previously acquired capabilities. Additional details are provided in Appendix H.

5.3 Ablation Studies

The full **TYLER** combines operator-conditioned latent token synthesis, budget-aware operator invocation, and the operator-anchored loss $\mathcal{L}_{\text{anch}}$. Table 2 ablates these components on Qwen3-4B by incrementally enabling Stage 1 and Stage 2, and by replacing typed operators or the learned router under comparable invocation settings. The Swap variant keeps the router, budget, and anchor loss fixed, but exchanges the synthesis paths of O_s and O_g at inference time.

Rows (a)–(e) show that Stage 1 provides the largest gain, while Stage 2 improves selective operator use. Invoking learned typed operators at all candidate boundaries raises the average accuracy from 56.56 to 63.38, with GPQA-Diamond improving from 16.16 to 39.90, indicating transfer beyond in-domain math traces. Learned routing and budget control maintain this gain, and adding $\mathcal{L}_{\text{anch}}$ further improves the average to 65.78, outperforming CoT by 9.22 points and Stage 1 by 2.40 points. This suggests that anchored credit assignment is useful for sparse operator-invocation decisions.

Rows (f)–(g) further show that the improvement is not merely due to latent capacity or access to candidate boundaries. Sharing operator-specific modules reduces the average to 62.95, while swapping O_s and O_g drops it to 60.44, indicating that the learned state-update and global-orientation operators play distinct roles. Rows (h)–(l) show that random or fixed invocation schedules remain inferior. Overall, these results indicate that **TYLER** must jointly learn *which* latent operator to invoke and *where* to invoke it.

5.4 Additional Analysis

We further analyze whether **TYLER** learns meaningful control over latent computation from two perspectives: whether the router allocates latent operators according to task difficulty and reasoning stage, and whether the typed operators induce differentiated latent computation. An efficiency analysis is provided in Appendix I.

Operator Specialization. We next study whether the three typed operators induce distinct functional roles. As shown in Figure 4(a), SoftCoT (Xu et al., 2025a) mainly occupies a mixed latent region, whereas the representations conditioned on O_g, O_s, and O_p form operator-aligned clusters in the same PCA space. This separation suggests that

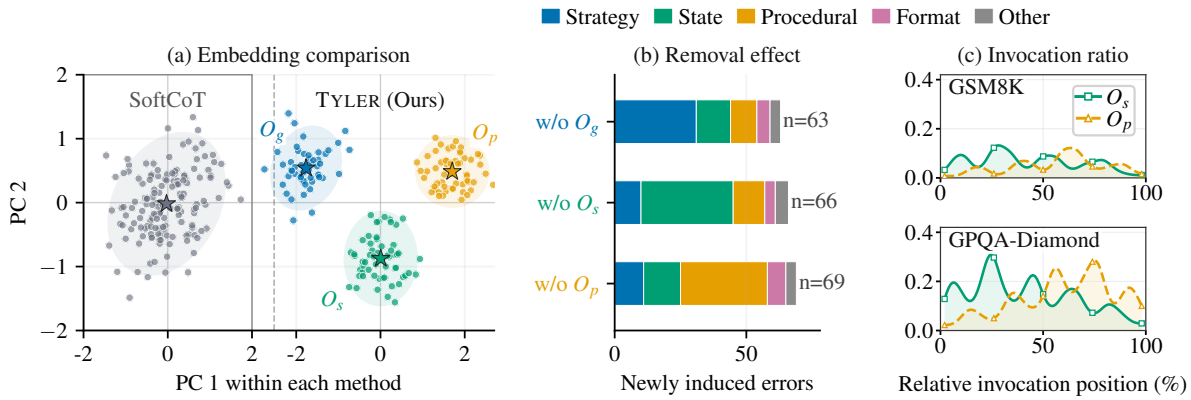

Figure 4: Diagnostics of typed latent operators with Qwen3-4B. (a) PCA visualization of latent representations under the same embedding protocol. (b) Error decomposition of newly induced failures after removing each typed operator on MATH-500. (c) Relative invocation-position distributions on GSM8K and GPQA.

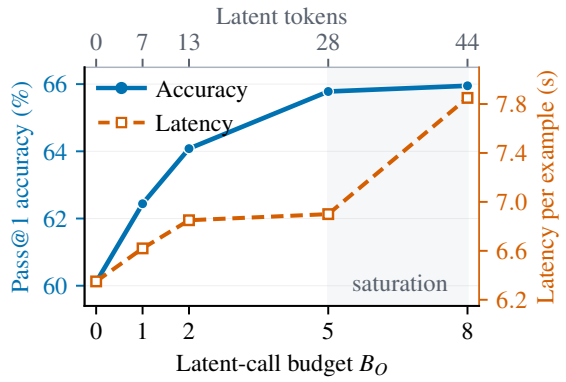

Figure 5: Latent-budget scaling on Qwen3-4B, averaged across GSM8K, MATH, GPQA, and TheoremQA.

the operators do not collapse into homogeneous latent representations. The removal analysis in Figure 4(b) further supports this interpretation: removing O_g leads to more strategy and setup errors, removing O_s increases state and arithmetic errors, and removing O_p produces more procedural and symbolic errors. Together, these results indicate that the typed operators learn complementary latent representations with distinct functional effects.

Task-Adaptive Invocation. Figure 4(c) reports the invocation ratios and relative positions of O_s and O_p on GSM8K and GPQA-Diamond. We visualize only O_s and O_p, since O_g is a global-orientation operator invoked at the beginning of generation. The router shows a clear task-adaptive pattern. On the relatively simple GSM8K task, the overall invocation ratio remains low, whereas GPQA-Diamond triggers latent operators more frequently, suggesting that the policy allocates more computation to harder reasoning tasks. The relative positions further reveal a stage-aware division of labor: O_s is invoked more often in the early and middle stages of reasoning, where local states and intermediate conclusions are actively maintained; O_p becomes more frequent in later stages, where reusable solution procedures are more likely to dominate. These results indicate that **TYLER** does not follow a rigid schedule, but learns to invoke latent computation according to both task complexity and reasoning state.

Token Budget Scaling. Figure 5 studies the effect of the latent-call budget. Accuracy improves rapidly as B_O increases from 0 to 5, showing that **TYLER** benefits from additional latent computation. However, when the budget increases further to $B_O = 8$, the marginal accuracy gain becomes very small, while latency increases noticeably. Thus, $B_O = 5$ provides a better accuracy–latency trade-off than using a larger budget.

6 Conclusion

We introduced **TYLER**, a framework that reframes latent reasoning as online, typed, and budgeted operator invocation. By combining three typed latent operators with a budget-aware policy, **TYLER** enables autoregressive decoding to decide when to invoke internal computation and which latent operation to perform under a computation budget. Across extensive experiments on three backbone LLMs, **TYLER** improves average performance by up to 14.49 points over CoT and by up to 4.30 points over the strongest competing baseline. It also generalizes across diverse reasoning domains

and achieves the best final-stage performance with the lowest forgetting in sequential adaptation.

7 Limitations

We note four limitations. First, our evaluation spans 1.5B–4B backbones across three model families; whether typed latent operators retain their advantage at substantially larger scales remains an open empirical question. Second, **TYLER** relies on rationale-like supervision to construct candidate boundaries in Stage 1. Although Stage 2 learns a routing policy over these candidates, fully unsupervised boundary discovery remains future work. Third, operator invocation triggers silent latent-token synthesis, which improves efficiency but reduces direct auditability compared with visible chain-of-thought rationales. The intended roles of the latent operators are learned rather than guaranteed by construction, so deployments should log operator identities, invocation positions, and budgets when traceability matters. Fourth, our experiments focus on reasoning benchmarks with verifier-style evaluation; broader open-ended generation tasks may require different rewards and additional safeguards against unfaithful or hard-to-inspect latent computation.

8 Ethical Considerations

This work studies latent reasoning as a mechanism for improving the efficiency and effectiveness of language-model reasoning. The experiments are conducted on public reasoning benchmarks and do not involve human subjects, private user data, or the collection of personally identifiable information.

The main potential risk is dual use: improving the reasoning capability or inference efficiency of language models may also reduce the cost of deploying models in harmful applications. In addition, latent computation may make intermediate reasoning less transparent than explicit CoT generation, which could complicate debugging and auditing. Our work is intended as a research contribution on controllable latent computation rather than a deployed system. We encourage future applications of this method to include standard safety evaluations, misuse analysis, and appropriate monitoring when used in real-world settings.

References

Elie Bakouch, Loubna Ben Allal, Anton Lozhkov, Nouamane Tazi, Lewis Tunstall, Carlos Miguel Patiño, Edward Beeching, Aymeric Roucher, Aksel Joonas Reedi, Quentin Gallouédec, Kashif Rasul, Nathan Habib, Clémentine Fourrier, Hynek Kydlicek, Guilherme Penedo, Hugo Larcher, Mathieu Morlon, Vaibhav Srivastav, Joshua Lochner, and 4 others. 2025. SmolLM3: smol, multilingual, long-context reasoner. https://huggingface.co/blog/smollm3.

Rich Caruana. 1997. Multitask learning. *Machine learning*, 28(1):41–75.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, and 1 others. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*.

Wenhu Chen, Ming Yin, Max Ku, Pan Lu, Yixin Wan, Xueguang Ma, Jianyu Xu, Xinyi Wang, and Tony Xia. 2023. Theoremqa: A theorem-driven question answering dataset. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7889–7901.

Xinghao Chen, Anhao Zhao, Heming Xia, Xuan Lu, Hanlin Wang, Yanjun Chen, Wei Zhang, Jian Wang, Wenjie Li, and Xiaoyu Shen. 2025. Reasoning beyond language: A comprehensive survey on latent chain-of-thought reasoning. *arXiv preprint arXiv:2505.16782*.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, and 1 others. 2021. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*.

Ronan Collobert and Jason Weston. 2008. A unified architecture for natural language processing: Deep neural networks with multitask learning. In *Proceedings of the 25th international conference on Machine learning*, pages 160–167.

Matthias De Lange, Rahaf Aljundi, Marc Masana, Sarah Parisot, Xu Jia, Aleš Leonardis, Gregory Slabaugh, and Tinne Tuytelaars. 2021. A continual learning survey: Defying forgetting in classification tasks. *IEEE transactions on pattern analysis and machine intelligence*, 44(7):3366–3385.

Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. 2024. Think before you speak: Training language models with pause tokens. In *International Conference on Learning Representations*, volume 2024, pages 27896–27923.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and 1 others. 2025. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nature*, 645(8081):633–638.

Shibo Hao, Sainbayar Sukhbaatar, DiJia Su, Xian Li, Zhiting Hu, Jason Weston, and Yuandong Tian. 2022. Training large language models to reason in a continuous latent space, 2024. *URL https://arxiv. org/abs/2412.06769*, 98.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. 2021. Measuring mathematical problem solving with the math dataset.

Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Liang Wang, Weizhu Chen, and 1 others. 2022. Lora: Low-rank adaptation of large language models. volume 1, page 3.

Hugging Face. 2025. Open r1: A fully open reproduction of deepseek-r1.

Aaron Jaech, Adam Kalai, Adam Lerer, Adam Richardson, Ahmed El-Kishky, Aiden Low, Alec Helyar, Aleksander Madry, Alex Beutel, Alex Carney, and 1 others. 2024. Openai o1 system card. *arXiv preprint arXiv:2412.16720*.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, and 1 others. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

Zhizhong Li and Derek Hoiem. 2017. Learning without forgetting. *IEEE transactions on pattern analysis and machine intelligence*, 40(12):2935–2947.

Weihao Liu, Dehai Min, and Lu Cheng. 2026. Latent thoughts tuning: Bridging context and reasoning with fused information in latent tokens. *arXiv preprint arXiv:2602.10229*.

Michael McCloskey and Neal J Cohen. 1989. *Catastrophic interference in connectionist networks: The sequential learning problem*, volume 24, pages 109–165. Elsevier.

Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, and 1 others. 2022. Training language models to follow instructions with human feedback. volume 35, pages 27730–27744.

Jacob Pfau, William Merrill, and Samuel R Bowman. 2024. Let's think dot by dot: Hidden computation in transformer language models. *arXiv preprint arXiv:2404.15758*.

Anastasia Razdaibiedina, Yuning Mao, Rui Hou, Madian Khabsa, Mike Lewis, and Amjad Almahairi. 2023. Progressive prompts: Continual learning for language models. *arXiv preprint arXiv:2301.12314*.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. 2023. Gpqa: A graduate-level google-proof q&a benchmark.

Sebastian Ruder. 2017. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*.

Victor Sanh, Albert Webson, Colin Raffel, Stephen H Bach, Lintang Sutawika, Zaid Alyafeai, Antoine Chaffin, Arnaud Stiegler, Teven Le Scao, Arun Raja, and 1 others. 2021. Multitask prompted training enables zero-shot task generalization.

Timo Schick, Jane Dwivedi-Yu, Roberto Dessì, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. 2023. Toolformer: Language models can teach themselves to use tools. volume 36, pages 68539–68551.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Zhihong Shao, Peiyi Wang, Qihao Zhu, Runxin Xu, Junxiao Song, Xiao Bi, Haowei Zhang, Mingchuan Zhang, YK Li, Yang Wu, and 1 others. 2024. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*.

Zhenyi Shen, Hanqi Yan, Linhai Zhang, Zhanghao Hu, Yali Du, and Yulan He. 2025. Codi: Compressing chain-of-thought into continuous space via self-distillation. pages 677–693.

Dachuan Shi, Abedelkadir Asi, Keying Li, Xiangchi Yuan, Leyan Pan, Wenke Lee, and Wen Xiao. 2025. Swireasoning: Switch-thinking in latent and explicit for pareto-superior reasoning llms.

Fan-Keng Sun, Cheng-Hao Ho, and Hung-Yi Lee. 2019. Lamol: Language modeling for lifelong language learning. *arXiv preprint arXiv:1909.03329*.

Xuezhi Wang, Jason Wei, Dale Schuurmans, Quoc Le, Ed Chi, Sharan Narang, Aakanksha Chowdhery, and Denny Zhou. 2022. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, and 1 others. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837.

Xilin Wei, Xiaoran Liu, Yuhang Zang, Xiaoyi Dong, Yuhang Cao, Jiaqi Wang, Xipeng Qiu, and Dahua Lin. 2025. Sim-cot: Supervised implicit chain-of-thought. *arXiv preprint arXiv:2509.20317*.

Yige Xu, Xu Guo, Zhiwei Zeng, and Chunyan Miao. 2025a. Softcot: Soft chain-of-thought for efficient reasoning with llms. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 23336–23351.

Yige Xu, Xu Guo, Zhiwei Zeng, and Chunyan Miao. 2025b. Softcot++: Test-time scaling with soft chain-of-thought reasoning. *arXiv preprint arXiv:2505.11484*.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and 1 others. 2025. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*.

An Yang, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chengyuan Li, Dayiheng Liu, Fei Huang, Haoran Wei, and 1 others. 2024. Qwen2.5 technical report.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. volume 36, pages 11809–11822.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. 2022. React: Synergizing reasoning and acting in language models.

Qiying Yu, Zheng Zhang, Ruofei Zhu, Yufeng Yuan, Xiaochen Zuo, Yu Yue, Weinan Dai, Tiantian Fan, Gaohong Liu, Lingjun Liu, and 1 others. 2026. Dapo: An open-source llm reinforcement learning system at scale. *Advances in Neural Information Processing Systems*, 38:113222–113244.

Guibin Zhang, Muxin Fu, and Shuicheng Yan. 2025. Memgen: Weaving generative latent memory for self-evolving agents. *arXiv preprint arXiv:2509.24704*.

Zhen Zhang, Xuehai He, Weixiang Yan, Ao Shen, Chenyang Zhao, and Xin Wang. 2026. Soft thinking: Unlocking the reasoning potential of llms in continuous concept space. *Advances in Neural Information Processing Systems*, 38:168990–169012.

Rui-Jie Zhu, Tianhao Peng, Tianhao Cheng, Xingwei Qu, Jinfa Huang, Dawei Zhu, Hao Wang, Kaiwen Xue, Xuanliang Zhang, Yong Shan, and 1 others. 2025. A survey on latent reasoning. *arXiv preprint arXiv:2507.06203*.

A Artifact

A.1 Artifact Licenses and Terms

We use publicly available datasets, benchmarks, open-weight models, and evaluation tools for research purposes only. We cite the original creators of all artifacts and follow their stated licenses or terms of use. Our use of these artifacts is consistent with their intended research use: benchmark datasets are used for training, validation, or evaluation as appropriate, and held-out benchmark splits are used only for evaluation. We do not redistribute benchmark data or model weights as part of this submission.

Any released artifacts will be limited to code, configuration files, and reproduction scripts. We will not release derivative datasets or redistribute model weights. These artifacts are intended only for research reproducibility and further academic study.

A.2 Artifact Documentation

Table 3 summarizes the main artifacts used in this work. All datasets and benchmarks are publicly available and are used for research purposes only. The language data used in our experiments is primarily English. The artifacts cover mathematical reasoning, scientific question answering, theorem-style reasoning, and code-generation tasks. They are not demographic or user-generated social datasets, and they are not used to study demographic groups, dialectal variation, or sociolinguistic phenomena. To the best of our knowledge, the benchmarks do not provide systematic demographic annotations for authors or represented populations; therefore, our analysis does not make demographic claims.

A.3 Dataset Statistics

Table 4 reports the statistics of the datasets and benchmarks used in this work. We use the official splits whenever available. For benchmarks without a training split in our experiments, we use them only for evaluation. We do not collect new natural-language data. Synthetic reasoning traces or latent-operator training instances are generated only from the corresponding public training data and are used for research purposes.

B Data Privacy and Content Safety

We do not collect new user data or use private, personally identifying, or sensitive information. All datasets used in this work are publicly available reasoning, science, theorem-proving, and code-generation benchmarks. These benchmarks are used only for research purposes, following their intended use and access conditions.

To check for privacy and content risks, we reviewed the dataset descriptions and manually in-

Artifact	Type	Domain	Language / Format
GSM8K	Dataset	Grade-school math reasoning	English text
MATH-500	Benchmark	Competition-level mathematics	English text / equations
GPQA-Diamond	Benchmark	Graduate-level science QA	English text
TheoremQA	Benchmark	Theorem-style reasoning	English text / equations
HumanEval	Benchmark	Code-generation evaluation	Python code and English prompts
SmolLM3-3B	Model	General-purpose language modeling	Text model
Qwen3-4B	Model	General-purpose language modeling	Text model

Table 3: Documentation of the main artifacts used in this work, including their type, domain coverage, and language or format.

Dataset / Benchmark	Train	Dev / Validation	Test / Eval
GSM8K	7,473	–	1,319
MATH-500	–	–	500
GPQA-Diamond	–	–	198
TheoremQA	–	–	737
HumanEval	–	–	164

Table 4: Statistics of the main datasets and benchmarks used in this work. Dashes indicate that the corresponding split is not used in our experiments.

spected representative examples from the training, validation, and evaluation data used in our experiments. We found no fields designed to identify individual people, such as names linked to real individuals, addresses, phone numbers, email addresses, account identifiers, or other personally identifying information. The benchmarks also do not target offensive, hateful, or abusive language generation. Therefore, no additional anonymization was required. We do not redistribute the benchmark data or model weights as part of this submission.

C Use of AI Assistants

We used AI writing assistance tools only for language polishing, grammar checking, and improving the clarity of author-written text. These tools were not used to generate research ideas, design the method, conduct experiments, analyze results, or produce conclusions. All AI-assisted edits were reviewed, verified, and revised by the authors, who take full responsibility for the content of the paper.

D Additional Related Works

Continual learning. Continual learning studies sequential adaptation under the risk of catastrophic interference, where learning a new task degrades performance on earlier tasks (McCloskey and Cohen, 1989; De Lange et al., 2021). Representative strategies include regularizing important parameters to preserve previous behavior (Kirkpatrick et al., 2017), distilling old-task predictions during new-task training (Li and Hoiem, 2017), replaying or generating samples from past tasks in lifelong language learning (Sun et al., 2019), and allocating lightweight prompt parameters across a task stream (Razdaibiedina et al., 2023). These methods primarily protect previously acquired behavior by constraining parameter updates, revisiting past data, or assigning task-specific capacity. Our evaluation is complementary: it does not introduce a general continual-learning algorithm, but asks whether typed latent operators can serve as a low-interference adaptation interface during sequential reasoning-task updates.

Multi-task learning. Multi-task learning improves transfer by training a shared model over multiple related objectives (Caruana, 1997; Ruder, 2017). In NLP, this idea has appeared in shared neural architectures for multiple language-processing tasks (Collobert and Weston, 2008) and, more recently, in large-scale prompted task mixtures that induce zero-shot generalization (Sanh et al., 2021). Unlike multi-task training, the continual-learning protocol in Appendix H exposes tasks sequentially rather than as a joint mixture. TYLER is orthogonal to both settings: instead of relying solely on a monolithic shared representation, it provides a small set of typed latent computation paths and learns when to route a reasoning trajectory through them. This makes the experiment a test of whether operator-mediated latent computation reduces interference while preserving cross-task transfer.

E Experiment Details

E.1 Training Data

We use 15K problem–solution traces sampled from OpenR1-Math (Hugging Face, 2025) to train **TYLER**. The same training pool is used for Stage 1 operator-conditioned latent-token synthesis and Stage 2 budgeted routing, while the backbone-specific SFT and GRPO baselines use the same data and prompt templates for controlled comparison. Evaluation benchmarks are used only for testing, and no test split is used for model selection.

E.2 Benchmarks and Metrics

We evaluate four reasoning benchmarks. GSM8K (Cobbe et al., 2021) and MATH-500 (Hendrycks et al., 2021) evaluate mathematical reasoning at different levels of symbolic complexity. GPQA-Diamond (Rein et al., 2023) evaluates graduate-level scientific reasoning. TheoremQA (Chen et al., 2023) evaluates theorem-oriented reasoning that requires applying formal concepts and structured solution patterns.

All main results report Pass@1 accuracy under greedy decoding. The macro-average is computed as the unweighted average over GSM8K, MATH-500, GPQA-Diamond, and TheoremQA. For all benchmarks, we require the model to wrap its final answer with `\boxed{}`. Outputs without a boxed final answer are judged as incorrect. For boxed outputs, we extract the content inside `\boxed{}` and apply the benchmark-specific answer normalizer or verifier.

E.3 Baselines and Budget Matching

We compare against explicit-reasoning baselines, including CoT (Wei et al., 2022), SFT (Ouyang et al., 2022), and GRPO (Shao et al., 2024), and latent-reasoning baselines, including SoftCoT (Xu et al., 2025a), Soft-Thinking (Zhang et al., 2026), MemGen (Zhang et al., 2025), and SwiReasoning (Shi et al., 2025). All baselines use the same backbone checkpoint, tokenizer, prompt template, and decoding setting whenever applicable. Baseline-specific hyperparameters follow the authors' recommended settings when available; otherwise, we tune them on the same development split used for **TYLER**.

E.4 Implementation Details

Table 5 summarizes the implementation configuration.

E.5 Result Reporting.

Unless otherwise specified, all results are reported as pass@1 accuracy on the official evaluation split of each benchmark. Each score is computed over all examples in the corresponding evaluation set, and the average score is the arithmetic mean across benchmarks. We report single-run results rather than the maximum over multiple random seeds or prompt trials. For training-based methods, we use a fixed random seed and keep the decoding and evaluation protocol identical across methods. We do not select the best result from repeated runs.

F Training Details

We train **TYLER** in two stages. Stage 1 learns typed latent operators while keeping the backbone LLM frozen. Stage 2 freezes the learned synthesizer and operator-specific components, and trains the backbone to decide *when* to invoke latent computation and *which* operator to select, under an explicit budget. Implementation hyperparameters are summarized in Appendix E.4.

F.1 Stage 1: Latent Synthesis Optimization

In Stage 1, the backbone LLM with parameters θ is frozen, and we optimize the latent synthesizer ϕ together with the operator-specific components $\{Q_k, \mathrm{Proj}_k\}_{O_k \in \mathcal{O}}$. For brevity, we collect all Stage 1 trainable parameters as $\Phi = (\phi, \{Q_k, \mathrm{Proj}_k\}_{O_k \in \mathcal{O}})$. For each supervised instance $(q, y_{1:T})$, we construct a set of candidate boundary–operator pairs $\mathcal{B}(q, y)$ by parsing the target sequence with structure-aware rules:

- O_g is associated with *answer starts*: the first position immediately following the question, and any position following solution-onset cues such as "Let me solve", "Solution:", or the analogous markers in code/math templates.
- O_s is associated with *reasoning-step boundaries*: positions following `\n\n` or enumerated step markers ("Step i:", "i.") detected by a lightweight regex parser.
- O_p is associated with *structural boundaries*: positions immediately before formula spans (`$...$`), fenced code blocks (```` ``` ````), or structured output schemata (e.g., JSON keys).

For each training step, we sample a pair $(b, k) \sim \mathcal{B}(q, y)$ and construct the prefix embedding sequence x_b from the question and the ground-truth prefix $y_{<b}$. The selected operator produces latent

Configuration	Parameter	Stage 1	Stage 2
Model	Backbones	Qwen2.5-1.5B-Instruct, SmolLM3-3B, Qwen3-4B	
	PEFT Method	LoRA	LoRA
	LoRA rank	8	8
	LoRA alpha	16	16
	LoRA dropout	0.1	0.1
	Target modules	`q_proj`, `v_proj`	`q_proj`, `v_proj`
Latent operators	Operator types	O_g, O_s, O_p	
	Synthesizer LoRA Parameters	Random initialization	Frozen from Stage 1
	Latent length N_g	8	8
	Latent length N_s	4	4
	Latent length N_p	4	4
Training	Training data	15K OpenR1-Math traces	
	Batch size	8	8
	Epochs / steps	2 epochs	1 epochs
	Learning rate	1×10^{-5}	1×10^{-5}
	Optimizer	AdamW	AdamW
	Scheduler	Cosine	Cosine
	Warmup ratio	0.1	0.1
	Random seeds	42	
GRPO-Relative	GRPO group size G	–	8
	Clip ratio ϵ	–	0.2
	KL coefficient β	–	0.03
	Latent-call budget B_O	–	5
	Budget penalty λ	–	0.1
	Anchor weight α	–	0.1
	Success threshold ρ_0	–	verifier success
Evaluation	Decoding	Greedy decoding	
	Max new tokens	2048	
	Answer extractor	`\boxed{}` parser with benchmark-specific normalization	

Table 5: Implementation configuration for **TYLER**. Stage 2 uses a per-trajectory latent-call budget of $B_O = 5$; task success is the benchmark-specific exact-match or verifier score.

tokens $z_b^k = O_k(x_b)$ via Eqs. (8)–(9), which are then appended to x_b following Eq. (7). We do not directly supervise z_b^k; instead, the latent tokens are trained only through their contribution to predicting subsequent visible tokens. Let $\tilde{c}_t$ denote the latent-augmented context preceding y_t (i.e., $\tilde{c}_t = (x_b, z_b^k, y_{b:t-1})$ for $t \geq b$, and the standard prefix otherwise). The sampled boundary only determines *where* latent tokens are appended; the loss is applied to the *entire* target sequence so that gradients flow back through z_b^k to Φ from every downstream position, preventing the latent module from overfitting to local suffix prediction:

$$\mathcal{L}_{\text{stage1}} = -\sum_{t=1}^{T} \log p_{\theta,\Phi}(y_t \mid \tilde{c}_t), \qquad (13)$$

where the gradient updates only Φ since θ is frozen.

F.2 Stage 2: Operator Invocation Optimization

Stage 2 learns a step-level operator invocation policy that decides whether to emit a visible token or to invoke a typed latent operator. We freeze Φ from Stage 1 and optimize the backbone parameters θ together with the extended head rows ψ introduced in §3.3.

Group-relative policy optimization. For each prompt q, the current policy samples a group of trajectories $\mathcal{G}(q) = \{\tau_i\}_{i=1}^{G}$. Each trajectory receives the same budget-aware reward used in the main text:

$$R(\tau) = R_{\text{task}}(\tau) - \lambda P_{\text{bud}}(\tau), \qquad (14)$$

where

$$P_{\text{bud}}(\tau) = s(\tau)(n_{\text{op}}(\tau) - B_O)_+. \qquad (15)$$

Here, $s(\tau) \in \{0, 1\}$ indicates task success, $n_{\text{op}}(\tau)$ is the number of latent-computation calls in τ, and B_O is the per-trajectory call budget. The penalty is non-zero only on *successful, over-budget* trajectories: failures contribute no budget penalty so that exploration on hard instances is not suppressed, and under-budget successes are free so that the model

has no incentive to remove useful latent computation.

Let $\bar{R}_q$ and σ_q denote the mean and standard deviation of rewards in $\mathcal{G}(q)$. GRPO assigns each trajectory a group-relative advantage

$$A_i = \frac{R(\tau_i) - \bar{R}_q}{\sigma_q + \epsilon}. \tag{16}$$

For any visible-token or operator-invocation action $a_{i,j}$ at context $x_{i,j}$, define the importance-sampling ratio as

$$r_{i,j} = \frac{p_{\theta,\psi}(a_{i,j} \mid x_{i,j})}{p_{\text{old}}(a_{i,j} \mid x_{i,j})}. \tag{17}$$

Let $\bar{r}_{i,j} = \text{clip}(r_{i,j}, 1-\epsilon, 1+\epsilon)$ denote the clipped ratio. The clipped per-action surrogate is

$$\ell_{i,j}^{\text{clip}} = \min(r_{i,j} A_i,\ \bar{r}_{i,j} A_i)\,. \tag{18}$$

The clipped GRPO objective (Shao et al., 2024; Guo et al., 2025) is

$$\begin{aligned}\mathcal{J}_{\text{GRPO}}(R) = \frac{1}{G}\sum_{i=1}^{G}\frac{1}{|\tau_i|}\sum_{j\in\tau_i}\ell_{i,j}^{\text{clip}} \\ - \beta\, D_{\text{KL}}(p_{\theta,\psi}\,\|\,p_{\text{ref}})\,,\end{aligned} \tag{19}$$

and the minimized GRPO loss is $\mathcal{L}_{\text{GRPO}}(R) = -\mathcal{J}_{\text{GRPO}}(R)$.

Operator-anchored supervision. The GRPO loss applies the same trajectory-level advantage across all actions in a sampled response. This is sufficient for task-level learning, but operator invocations are sparse relative to visible-token actions, so their selection gradients can be diluted. We therefore add an operator-anchored auxiliary objective that applies the same group-relative advantage directly to operator invocations.

For each sampled trajectory τ_i, let

$$M_{\mathcal{O}}(\tau_i) = \{\, j \mid a_{i,j} \in \mathcal{O} \,\} \tag{20}$$

be the set of positions where the policy invokes a latent operator, and let $N_{\mathcal{O}} = \sum_i |M_{\mathcal{O}}(\tau_i)|$. The auxiliary loss is

$$\mathcal{L}_{\text{anch}} = -\frac{1}{N_{\mathcal{O}}}\sum_{i=1}^{G}\sum_{j\in M_{\mathcal{O}}(\tau_i)}\ell_{i,j}^{\text{clip}}. \tag{21}$$

When $N_{\mathcal{O}} = 0$, we set $\mathcal{L}_{\text{anch}} = 0$. This term reuses the GRPO advantage rather than introducing extra intervention rollouts, and concentrates additional gradient on the operator invocations that trigger typed latent operators.

Combined objective. The Stage 2 loss combines the standard GRPO loss with the operator-anchored auxiliary term:

$$\min_{\theta,\psi}\ \mathcal{L}_{\text{stage2}} = \mathcal{L}_{\text{GRPO}}(R) + \alpha\,\mathcal{L}_{\text{anch}}, \tag{22}$$

where α controls the auxiliary weight. We tune $(\lambda, B_O, \alpha, G)$ on the development split.

G Results on Qwen2.5-1.5B-Instruct

We report the full evaluation on Qwen2.5-1.5B-Instruct (Yang et al., 2024) in this appendix because the 1.5B regime exposes several small-model failure modes for prior latent-reasoning methods that are absent or weaker on the SmolLM3-3B/Qwen3-4B backbones used in the main paper. Setup, baselines, and metrics follow §5.1; all latent baselines are budget-matched to within $\pm 10\%$ of **TYLER**'s total visible + latent token count, and all numbers are Pass@1 under greedy decoding.

Small-model regime exposes latent-baseline brittleness. At 1.5B, three patterns emerge that motivate the design of **TYLER**. First, the *implicit* latent baselines do not transfer down: SoftCoT collapses to 31.62 on the macro-average (-6.55 vs. CoT), and Soft-Thinking matches CoT on average but loses 6.06 points on GPQA-Diamond. This is consistent with the original SoftCoT report (Xu et al., 2025a) that the assistant-generated soft thoughts require a sufficiently expressive backbone to be consumed productively. Second, the *hybrid* baselines are uneven: MemGen becomes the strongest prior latent baseline with a $+2.89$ macro-average gain over CoT, but entropy-based switching with SwiReasoning still falls below CoT. Third, RL alone (GRPO, $+1.99$ over CoT) wins GSM8K, MATH-500, and TheoremQA among prior baselines, but its GPQA-Diamond accuracy (12.12) regresses below CoT, suggesting that the policy improvements concentrate on the mathematical subset on which the reward is trained.

TYLER dominates the macro-average and the knowledge-intensive split. **TYLER** (Stage 1) attains a strong macro-average (41.65), exceeding the strongest prior baseline MemGen by $+0.59$ points and CoT by $+3.48$. On GPQA-Diamond, **TYLER** reaches 21.21, matching MemGen and improving over GRPO by $+9.09$ points. On the remaining three benchmarks **TYLER** is within 0.40–1.80 points of the best prior baseline. The Stage 2 row further raises the average to 44.05 after adding

Method	GSM8K	MATH-500	GPQA-Diamond	TheoremQA	Average	Δ_{CoT}
CoT	64.67	52.80	17.17	18.07	38.17	–
SFT	66.49	47.20	19.70	15.93	37.33	−0.84
GRPO	<u>74.30</u>	<u>54.80</u>	12.12	<u>19.41</u>	40.16	+1.99
SoftCoT	51.55	41.80	19.19	13.92	31.62	−6.55
Soft-Thinking	68.92	51.60	11.11	16.87	37.13	−1.04
MemGen	71.19	53.20	<u>21.21</u>	18.62	41.06	+2.89
SwiReasoning	61.71	43.60	11.11	17.62	33.51	−4.66
TYLER (Ours)						
+ Stage 1	73.39	53.00	<u>21.21</u>	19.01	<u>41.65</u>	<u>+3.48</u>
+ Stage 2	**75.28**	**57.60**	**23.23**	**20.08**	**44.05**	**+5.88**

Table 6: Pass@1 (%, greedy decoding) on Qwen2.5-1.5B-Instruct. **Best** and <u>second-best</u> among complete rows; Δ_{CoT} is the macro-average gain over CoT. All latent baselines budget-matched to within $\pm 10\%$ of **TYLER**.

budget-aware routing and operator-anchored supervision.

H Continual-Learning Evaluation

This appendix presents a fixed-order continual-learning evaluation to examine whether synthetic latent tokens provide a low-interference interface for sequential adaptation. We use Qwen2.5-1.5B-Instruct as the backbone and adapt each method sequentially across four task families: code-style reasoning evaluated on HumanEval (Chen et al., 2021) → GPQA-style science reasoning → GSM8K arithmetic reasoning → TheoremQA-style theorem reasoning. After each adaptation stage, the resulting model is evaluated on all four benchmarks, including both the newly adapted domain and previously observed domains. For each stage, adaptation uses training or development traces from the corresponding task family, while the held-out benchmark splits are used exclusively for evaluation. This setup provides a controlled evaluation of sequential interference under a fixed task order, rather than an exhaustive continual-learning benchmark over all possible task permutations.

We report the macro-average over the four evaluation benchmarks and the stage-wise forgetting score:

$$\mathcal{F}_t = \frac{1}{|\mathcal{S}_t|} \sum_{i \in \mathcal{S}_t} \left(\max_{s \le t} A_{s,i} - A_{t,i} \right), \qquad (23)$$

where $\mathcal{S}_t$ denotes the set of task families observed up to stage t, and $A_{t,i}$ is the accuracy on task family i after the t-th adaptation stage. Lower values of $\mathcal{F}_t$ indicate stronger retention of previously acquired capabilities. Since the newly introduced task at a given stage contributes zero forgetting by construction, the forgetting score at stage 1 is trivially zero for all methods.

Improved Adaptation under Sequential Training. Across the four-stage adaptation sequence, **TYLER** achieves the highest macro-average after every stage. Its advantage over the strongest baseline increases from $+1.40$ points after code-style adaptation to $+3.48$ points after the final TheoremQA-style adaptation, where **TYLER** reaches 39.83 compared with 36.35 for MemGen, 34.08 for GRPO, and 28.23 for SFT. Under this fixed task order, the performance gap becomes larger as more adaptation stages are applied, suggesting that **TYLER** maintains stronger overall performance during sequential adaptation.

Reduced Forgetting on Previously Observed Tasks. **TYLER** also obtains the lowest forgetting score throughout the sequence. At stage 2, it achieves $\mathcal{F} = 1.15$, slightly below MemGen at 1.35 and below GRPO and SFT at 1.85 and 3.85, respectively. The gap becomes more pronounced in later stages. After the final adaptation stage, **TYLER** obtains $\mathcal{F} = 2.27$, reducing forgetting by 1.36 points relative to MemGen, 4.21 points relative to GRPO, and 9.63 points relative to SFT. These results suggest that operator-based latent adaptation can reduce interference with previously acquired capabilities compared with direct SFT or policy optimization alone.

A Better Stability–Plasticity Trade-Off. After the final TheoremQA-style adaptation, **TYLER**

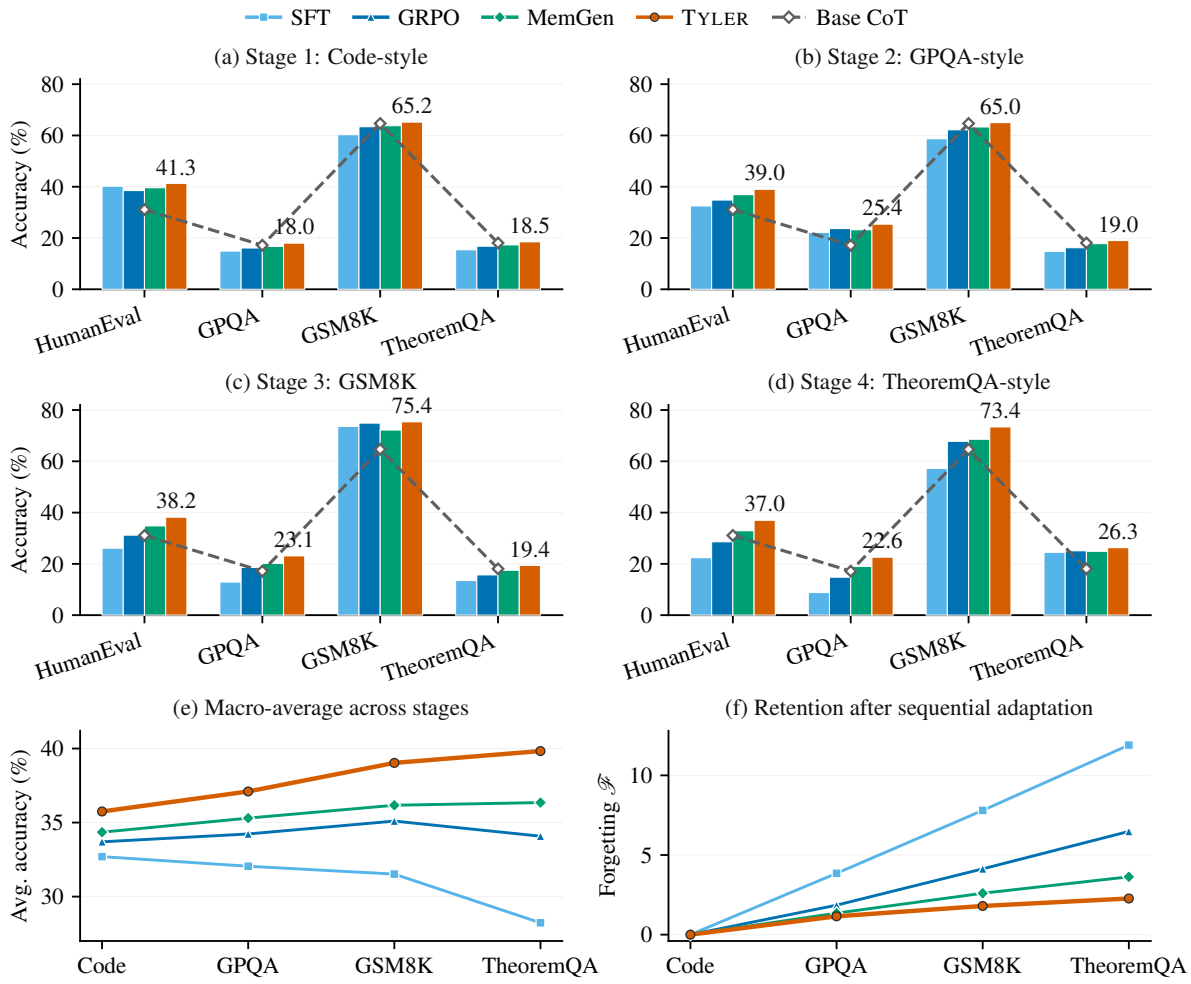

Figure 6: Fixed-order continual-learning evaluation on Qwen2.5-1.5B-Instruct. Panels (a)–(d) show benchmark performance after each sequential adaptation stage, with tasks on the horizontal axis and methods grouped by color. The dashed gray line denotes the base Vanilla CoT model. Panels (e)–(f) summarize the macro-average and the stage-wise forgetting score $\mathcal{F}$. Lower $\mathcal{F}$ indicates stronger retention of previously observed task capabilities.

achieves the best score on all four benchmarks: 37.00 on HumanEval, 22.60 on GPQA, 73.40 on GSM8K, and 26.30 on TheoremQA. Compared with MemGen, the gains are $+4.10$, $+3.60$, $+4.80$, and $+1.40$ points, respectively. SFT shows substantial improvement on the currently adapted TheoremQA-style task but suffers severe degradation on previously adapted domains, leading to the highest final forgetting score of 11.90. GRPO and MemGen mitigate this degradation to some extent, while **TYLER** achieves both the highest final macro-average and the lowest final forgetting score. This indicates a stronger stability–plasticity trade-off in the tested sequential adaptation setting.

I Computational Efficiency Analysis

I.1 Experimental Settings

We report end-to-end per-example inference time together with task performance. All methods are evaluated with greedy decoding, batch size 1, identical prompt templates, the same maximum generation length, and the same Qwen3-4B backbone. Latency is measured after tokenization and excludes data loading, answer extraction, verifier execution, and logging. For each method, we record visible generated tokens and latent tokens. The total token budget is the sum of visible and latent tokens, so latent computation is accounted for explicitly rather than treated as free computation.

Stage↓	Adapted Domain	Method	HumanEval	GPQA	GSM8K	TheoremQA	Avg.	$\mathcal{F}\downarrow$
–	Base	Vanilla CoT	31.10	17.17	64.67	18.07	32.75	–
1	HumanEval	SFT	40.20	14.90	60.30	15.40	32.70	0.00
		GRPO	38.50	16.10	63.40	16.80	33.70	0.00
		MemGen	39.60	16.70	63.80	17.30	34.35	0.00
		TYLER (Ours)	41.30	18.00	65.20	18.50	35.75	0.00
2	GPQA	SFT	32.50	22.20	58.70	14.80	32.05	3.85
		GRPO	34.80	23.70	62.20	16.20	34.23	1.85
		MemGen	36.90	23.20	63.30	17.80	35.30	1.35
		TYLER (Ours)	39.00	25.40	65.00	19.00	37.10	1.15
3	GSM8K	SFT	26.10	12.90	73.60	13.50	31.52	7.80
		GRPO	31.20	18.60	74.90	15.70	35.10	4.13
		MemGen	34.80	20.20	72.20	17.50	36.17	2.60
		TYLER (Ours)	38.20	23.10	75.40	19.40	39.03	1.80
4	TheoremQA	SFT	22.40	8.80	57.20	24.50	28.23	11.90
		GRPO	28.60	14.80	67.80	25.10	34.08	6.48
		MemGen	32.90	19.00	68.60	24.90	36.35	3.63
		TYLER (Ours)	37.00	22.60	73.40	26.30	39.83	2.27

Table 7: Fixed-order continual-learning evaluation on Qwen2.5-1.5B-Instruct. The downward arrow in the Stage column indicates the top-to-bottom sequential adaptation order. The macro-average is computed over all four evaluation benchmarks at every stage. The forgetting score $\mathcal{F}$ is computed over task families observed up to the current stage, with lower values indicating better retention.

I.2 Accuracy–Latency Tradeoff

Figure 7 summarizes the accuracy–efficiency tradeoff across GSM8K, MATH-500, TheoremQA, and GPQA-Diamond. The left panel reports macro-average Pass@1 against per-example latency, with marker area proportional to the total generated-token budget. The right panel decomposes the same budget into visible and latent tokens.

TYLER reaches the highest average accuracy while staying within a compact compute budget. Compared with CoT, **TYLER** improves average Pass@1 from 56.56 to 65.78 while reducing latency from 8.42s to 6.90s and reducing the total token budget from 1050.1 to 628.4 tokens. Compared with the strongest prior baseline by accuracy, GRPO, **TYLER** gains +3.13 points with a modest 0.32s latency increase. Compared with MemGen, **TYLER** is both more accurate and faster: it improves average Pass@1 by +4.50 points, reduces latency by 0.30s, and uses fewer total tokens (628.4 vs. 652.4).

I.3 Per-Benchmark Overhead

Table 8 breaks down **TYLER** latency by benchmark, ordered from easier to harder tasks. The total latency increases with task difficulty, from GSM8K to GPQA-Diamond, while the wall-clock time spent in latent synthesis remains a small fraction of the overall inference time.

Benchmark	Total (s)	Synth. (s)	Synth. Share (%)
GSM8K	4.90	0.38	7.8
MATH	6.30	0.49	7.8
Theorem	7.49	0.54	7.2
GPQA	8.90	0.58	6.5

Table 8: Per-benchmark latency breakdown for **TYLER** on Qwen3-4B. *Synth.* denotes wall-clock time spent in latent reasoning.

The synthesis overhead remains stable across benchmarks: latent reasoning accounts for 6.5–7.8% of total inference time. The larger latency on TheoremQA and GPQA-Diamond is therefore driven primarily by longer visible reasoning traces rather than an uncontrolled growth in the latent synthesis module. This behavior matches the intended design of **TYLER**: the router allocates a small number of typed latent computations while the base decoder remains responsible for producing the final visible answer.

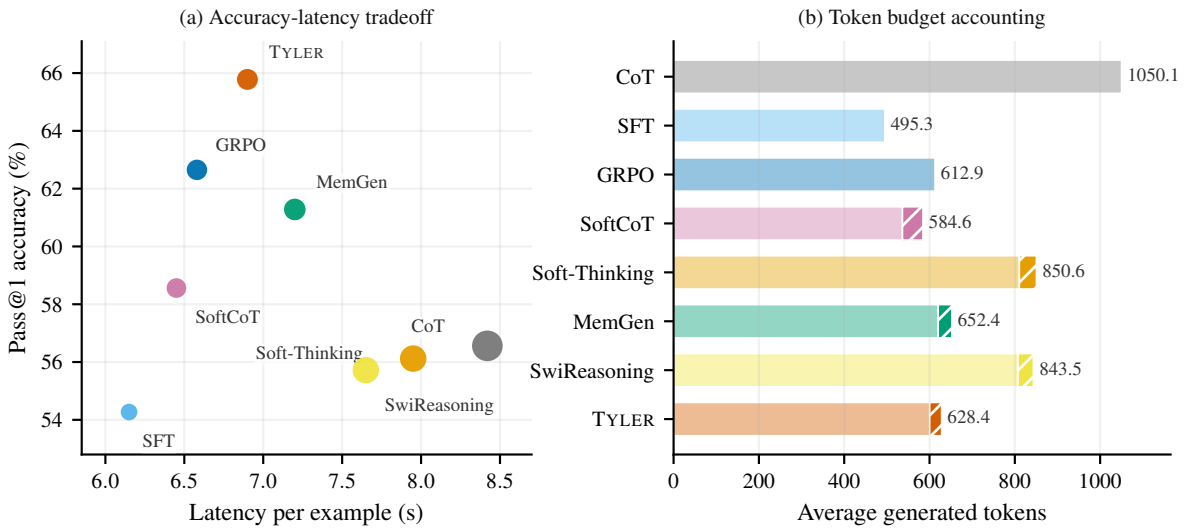

Figure 7: Computational efficiency on Qwen3-4B, averaged across GSM8K, MATH-500, TheoremQA, and GPQA-Diamond. Each method uses a fixed color across panels. In the token accounting panel, lighter bar segments denote visible tokens and hatched saturated caps denote latent tokens.

B_O	**Accuracy (%)**	**Latency (s)**	**Latent Tokens**
0	60.11	6.35	0
1	62.44	6.62	7
2	64.08	6.85	13
5	65.78	6.90	28
8	65.95	7.85	44

Table 9: Budget sweep for **TYLER** on Qwen3-4B, averaged across GSM8K, MATH-500, TheoremQA, and GPQA-Diamond.

I.4 Budget Sweep

Table 9 varies the latent budget on Qwen3-4B. The sweep tests whether **TYLER** benefits from additional latent computation and whether the gains saturate after a small budget.

Most of the gain appears under a small latent budget. Moving from $B_O = 0$ to $B_O = 2$ improves accuracy by $+3.97$ points with only 0.50s additional latency. The full budget $B_O = 5$ further improves accuracy to 65.78, yielding a $+5.67$ point gain over no latent routing. Increasing the budget to $B_O = 8$ adds 16 latent tokens and 0.95s latency but improves accuracy by only 0.17 points, indicating that the accuracy benefit saturates once the router has enough budget to invoke the most useful latent operators.