TEST-TIME REVISION: LATENT REASONING WITH EXECUTION IN THE LOOP

Anonymous authors
Paper under double-blind review

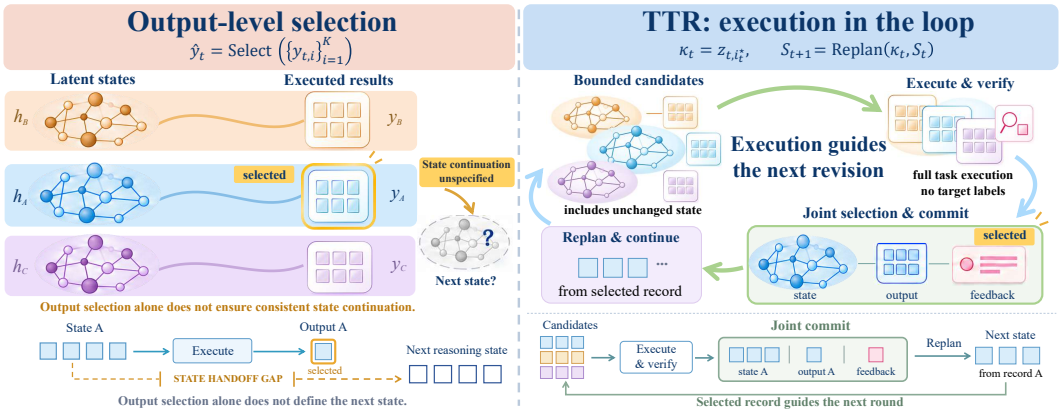


Figure 1: Output-level selection chooses among completed predictions but does not specify which latent state should guide the next round. TTR executes bounded state proposals, verifies their outputs, and jointly commits the selected state and output for subsequent reasoning.

ABSTRACT

Extra test-time computation improves predictions. However, updating hidden states during inference may improve intermediate scores yet still produce incorrect or invalid outputs after full task execution. Reranking can select a better output, but selection alone does not ensure that subsequent reasoning builds on the corresponding internal state. We introduce *Test-Time Revision (TTR)*, an approach to latent reasoning guided by task execution. TTR evaluates bounded changes to the current state through full task execution and jointly commits the selected output and its corresponding internal state. This preserves consistency between the selected output and the state used for subsequent reasoning, while allowing execution feedback to guide further revision. The same mechanism applies to Sudoku token states, Maze spatial representations, and Qwen prompt embeddings. Each task retains its own adapter, executor, and evaluation rules, while the original backbones remain frozen. TTR achieves strong performance on held-out Sudoku, Maze, and MATH500 data, solving 96.3% of Sudoku boards correctly. Matched Maze controls show that reusing the selected state improves subsequent execution, and the anonymous code is available at .

1 INTRODUCTION

Test-time computation methods improve predictions by allocating additional computation after receiving an input. Figure 1 contrasts output-level selection with continuing from the selected latent state. Recurrent models and models with adaptive depth repeatedly update internal states (Dehghani et al., 2019, Banino et al., 2021, Wang et al., 2025, Lu et al., 2026). Test-Time Training updates a hidden-state model within the test sequence (Sun et al., 2024), while iterative energy or constraint methods repeatedly improve candidate solutions (Du et al., 2024, Xu et al., 2025). Test-time control and voting explore alternative predictions (Wang et al., 2026, Kubo et al., 2026, Wang et al., 2022).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

However, an update to a hidden state may improve an intermediate score yet produce an incorrect or invalid output after full task execution. Reranking can compare completed outputs, but selection alone does not determine which internal state is used for subsequent reasoning. If the next round resumes from a different state, it may fail to build on the selected result. Self-Refine and Reflexion already use critiques, previous answers, or task feedback to guide subsequent attempts (Madaan et al., 2023, Shinn et al., 2023). These methods provide feedback through text, without explicitly comparing candidate latent states through their execution results. We therefore evaluate latent changes through full task execution and retain the selected state with its result, so each revision proceeds from an execution-validated record.

The challenge is to connect the evaluation of a complete result with the state used for further reasoning. For example, a Sudoku update may increase confidence in individual cells while producing a board that violates the constraints. Checking the complete board can reveal this failure, but the evaluation must also inform subsequent state updates. We therefore ask: *how can full task execution guide both the evaluation of latent states and subsequent reasoning from the selected state?*

To address this question, we introduce Test-Time Revision (TTR), an approach that uses task execution as feedback for latent reasoning. Our central idea is to connect two decisions: which state update to accept, and how reasoning should proceed from that update. TTR uses completed task results to inform both decisions, integrating evaluation and revision into a single feedback loop.

The same mechanism applies to Sudoku token states, Maze spatial representations, and Qwen prompt embeddings. Each task retains its own adapter, executor, evaluation rules, and procedure for preparing the next round. A new task supplies these components while reusing candidate tracking, the unchanged fallback, joint commit, and loop control. TTR is complementary to online Test-Time Training. The native backbones remain frozen, and the learned proposal parameters are frozen during inference, while execution feedback updates the state used for subsequent reasoning.

On held-out data, TTR achieves 96.3% Sudoku board accuracy, compared with 90.8% for TTC. On MATH500, it reaches 37.4% exact match versus 25.6% for Transformer under the same ten-generation budget. TTR also achieves 19.9% Maze strict branch-free accuracy. Sudoku uses one revision, Maze and MATH500 use two. Matched Maze controls show that reusing the selected state improves subsequent execution. Our contributions are:

- We introduce TTR, a method that evaluates proposed changes to latent states through full task execution. This allows complete task results to inform state selection, addressing the limitations of relying on intermediate scores alone.
- We connect output selection with subsequent reasoning by jointly retaining the selected output and its corresponding internal state. The selected state and execution feedback guide further revision, preserving consistency between the chosen result and the state used for the next revision round within the same native execution path.
- We evaluate TTR on Sudoku, Maze, and MATH500 with frozen backbones, showing improvements over Transformer, TTC, and TTT-Linear on the primary task metrics. Matched controls on Maze further demonstrate that continuing from the selected state improves subsequent execution under the same revision budget.

2 Related Work

Additional inference computation refines hidden states or searches over alternative candidates. Adaptive-depth models, Test-Time Training, Coconut, and Test-Time Control update or plan over latent states (Dehghani et al., 2019, Banino et al., 2021, Wang et al., 2025, Sun et al., 2024, Hao et al., 2024, Wang et al., 2026). Energy-based methods refine candidates with learned objectives (Du et al., 2024, Gladstone et al., 2026). TTR shares the use of extra computation, but evaluates every candidate through the native task and retains the state paired with the selected completed result.

Output methods vote or revise completed predictions. Chain-of-thought and self-consistency generate multiple paths (Wei et al., 2022, Wang et al., 2022). Tree of Thoughts searches intermediate paths (Yao et al., 2023), and ReAct interleaves reasoning with actions (Yao et al., 2022). C-Voting, Self-Refine, Reflexion, and verifier methods select outputs or supervise steps (Kubo et al., 2026, Madaan et al.,

2023, Shinn et al., 2023, Cobbe et al., 2021, Lightman et al., 2023). AlphaZero and MuZero plan with execution (Silver et al., 2017, Schrittwieser et al., 2020).

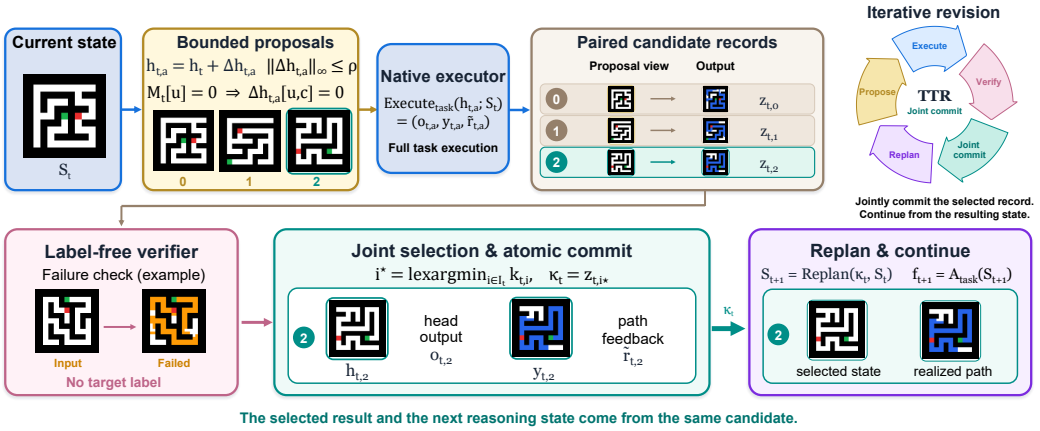


Figure 2: TTR constructs bounded latent proposals from the current task state, executes every candidate with the native executor, verifies the completed outputs, and atomically commits the selected output together with its latent state and realized feedback. The committed record is replanned into the next state, so execution guides both selection and subsequent reasoning. Task-specific adapters and executors vary across Sudoku, Maze, and MATH500, while the proposal, commit, and revision loop remains shared. All proposal parameters are frozen during inference.

3 METHODOLOGY

3.1 MOTIVATION: EXECUTION IN THE LOOP

A latent refiner changes an internal state before observing its complete task consequence, whereas an output reranker observes predictions without changing their generating state. TTR joins these paths by proposing bounded branches, executing them natively, and hard-committing one aligned latent/output/result tuple used to construct the next state. The unchanged incumbent remains a fallback. Task adapters, executors, and verifiers provide task-specific semantics, while proposal parameters remain frozen at evaluation. Figure 2 summarizes the shared execution–feedback contract.

3.2 PROBLEM FORMULATION

Let x be a task input and $y^\star$ its target, unavailable during inference. The frozen native system with parameters θ initializes a continuous state $h_0 = F_\theta(x)$, where F_θ is the native state encoder. The layer uses J learned branches per round and at most K revision rounds, indexed by $t \in \{0, \dots, K-1\}$. With batch size B and native channel dimension D, the current state is

$$S_t = (h_t, M_t, C_t, r_t), \tag{1}$$

where $h_t \in \mathbb{R}^{B\times\cdots\times D}$ is a channel-last latent, the omitted axes index token or spatial positions. M_t has one binary entry per sample and position, with one permitting revision. C_t stores input information and task context and is read-only within a round. r_t stores explicit realized feedback when used, while r_0 is task-initialized or absent. Qwen instead encodes draft feedback in h_t and C_t. Batch indices are suppressed unless shown explicitly. Selection and state updates are applied independently to each sample so that candidate records never cross sample boundaries.

Executing h_t returns a native head output o_t, complete task output y_t, and newly realized state $\widetilde{r}_t$:

$$(o_t, y_t, \widetilde{r}_t) = \mathrm{Execute}_{\mathrm{task}}(h_t, M_t, C_t, r_t). \tag{2}$$

The executor includes the complete Sudoku decoder, Maze path rasterization, or native Qwen generation. Its notation suppresses the configured decoding randomness.

Each task supplies four maps: a feature adapter A_{task}, an executor, a verifier, and a state update function REPLAN for the shared revision loop:

$$\begin{aligned} f_t &= A_{\text{task}}(S_t), & E_t &= \text{Execute}_{\text{task}}(H_t, S_t), \\ V_t &= \text{Verify}_{\text{task}}(\mathcal{P}_t, S_t), & S_{t+1} &= \text{Replan}_{\text{task}}(\kappa_t, S_t). \end{aligned} \tag{3}$$

Here f_t is adapted task evidence with the same shape as h_t, H_t is the candidate latent bank, and E_t contains its native execution results. $\mathcal{P}_t$ is the active candidate pool, V_t its ordered verifier keys, and κ_t the selected latent/output/realized-state record. The following subsections define candidate construction, selection, state updates, and the shared revision loop.

A verifier may inspect x, r_t, and candidate outputs, but not $y^\star$. The interface requires:

1. every candidate uses the same native executor as the incumbent,
2. every candidate bank contains an unchanged latent incumbent,
3. the selected latent, output, and realized state are passed atomically to replanning, never as a soft mixture, and
4. changing J or K changes inference computation only, not data roles or label access.

3.3 BOUNDED LATENT PROPOSALS

The shared TTR revision layer maps (h_t, f_t, M_t) to a candidate bank $H_t = (h_{t,a})_{a=0}^{J}$, an ordered bank retaining duplicate entries. Index $a = 0$ denotes the incumbent, $a = 1, \ldots, J$ enumerates learned branches in a fixed order over pairs (g, j), where g indexes a proposal group and j a branch within that group. Let Gate_g combine the latent and adapted task evidence, $\mathcal{D}_g$ be a learned down-projection, and Branch_g be a branch-conditioned low-rank up-projection. With e_j^g denoting a learned branch embedding, the shared proposal core is

$$q_t^g = \mathcal{D}_g(\text{Gate}_g(h_t, f_t)), \qquad \widehat{\Delta h}_{t,j}^{\,g} = \text{Branch}_g(q_t^g, e_j^g), \tag{4}$$

where q_t^g has the batch and position axes of h_t and d_{rank} channels. The exact task-specific parameterizations are given in Appendix C.1. The final proposal uses a residual coordinate bound $\rho \geq 0$:

$$\Delta h_{t,j}^g = \rho\, M_t \odot \gamma_j^g\, \widehat{\Delta h}_{t,j}^{\,g}, \qquad h_{t,j}^g = h_t + \Delta h_{t,j}^g, \tag{5}$$

where $\gamma_j^g \in [0, 1]$ is the clamped branch scale, or one when scaling is disabled. The product $\odot$ is elementwise and broadcasts M_t over channels. The parameterizations used here apply a final $\tanh$ to the branch direction, so each entry of $\widehat{\Delta h}_{t,j}^{\,g}$ lies in $[-1, 1]$. Thus each residual coordinate has magnitude at most ρ and is zero at immutable positions. Also,

$$h_{t,0} = h_t, \qquad \Delta h_{t,0} = 0. \tag{6}$$

For the learned index a assigned to (g, j), set $h_{t,a} = h_{t,j}^g$ and $\Delta h_{t,a} = \Delta h_{t,j}^g$. The incumbent remains in the bank. Proposal groups have separate down/up projections and branch embeddings.

3.4 EXECUTION, VERIFICATION, AND HARD COMMIT

Latent candidates are not final predictions. Every candidate in the current bank first passes through the task's frozen native computation:

$$E_t = \text{Execute}_{\text{task}}(H_t, S_t) = \big((o_{t,a}, y_{t,a}, \widetilde{r}_{t,a})\big)_{a=0}^{J}, \tag{7}$$

Equation 7 applies Equation 2 to each $h_{t,a}$, with M_t, C_t, and r_t shared across candidates. Here $o_{t,a}$ is a native head output, $y_{t,a}$ is a complete output, and $\widetilde{r}_{t,a}$ is the newly realized state of candidate a. Associate each execution with its source latent as $\mathcal{B}_t = \big((h_{t,a}, o_{t,a}, y_{t,a}, \widetilde{r}_{t,a})\big)_{a=0}^{J}$. The active pool $\mathcal{P}_t$ is this bank for per-round selection or an archive for global selection. Let $I_t = \{0, \ldots, |\mathcal{P}_t| - 1\}$ be the flattened pool-index set and write its ith tuple as $z_{t,i} = (\bar{h}_{t,i}, \bar{o}_{t,i}, \bar{y}_{t,i}, \bar{r}_{t,i})$. Bars reindex a candidate record in the pool, possibly from an earlier round. $\bar{r}_{t,i}$ is that candidate's $\widetilde{r}$, not the prior r_t. The verifier returns a scalar score (a one-term key) or a lexicographic key $k_{t,i}$ for each $i \in I_t$,

computed without the target label, with smaller keys preferred. For sample b, write these records and keys as $z_{t,b,i}$ and $k_{t,b,i}$. TTR chooses

$$i^\star_{t,b} = \operatorname*{lexargmin}_{i \in I_t} k_{t,b,i}, \qquad \kappa_{t,b} = z_{t,b,i^\star_{t,b}}, \qquad \kappa_t = (\kappa_{t,b})_{b=1}^{B}. \tag{8}$$

The argmin is per sample and selects the smallest pool index among candidates with the minimum key. In a per-round pool, this retains the incumbent when it ties for the best key. TTR returns the selected latent, output, complete result, and realized state together. This committed tuple drives replanning for the next revision rather than a soft candidate average.

3.5 State Update and Iterative Revision

After selection, $\mathrm{Replan}_{\mathrm{task}}$ uses κ_t and S_t to construct S_{t+1}, as in Equation 3. This corresponds to the Task replan and Next state modules in Figure 2. Maze retains the selected hidden map and realized path. Sudoku re-encodes the selected board, Qwen re-encodes the selected draft with the original question and rebuilds the prompt context and mask. Thus the next proposal uses the state derived from the selected record.

At the next round, $f_{t+1} = A_{\mathrm{task}}(S_{t+1})$ provides updated task evidence for further proposals. MATH500 also retains earlier candidate records in a separate archive for selection. Only the selected draft and its task context are used to construct the next prompt. The complete loop is

$$S_t \to H_t \to E_t \to \mathcal{P}_t \to V_t \to \kappa_t \to S_{t+1}. \tag{9}$$

Further rounds run while the inference budget and configured stopping rule permit. When revision stops, the selected complete output is returned to the task evaluator.

3.6 Training Objective and Optimization Boundary

The current experiments freeze all native backbone and head parameters. Only the TTR layer and, when configured, its task feature adapter are trainable. At inference, these parameters are also frozen. Revision updates the task state. For scalar branch losses $\ell_{b,a}$ computed with fit labels, b indexes a sample and a uses the candidate ordering above. The training state index is suppressed. Let $\mathcal{J}_{\mathrm{fit}} \subseteq \{1, \dots, J\}$ be the nonempty set of learned branches enabled for fitting. The fit index set is $\mathcal{T}_b = \mathcal{J}_{\mathrm{fit}}$ for Sudoku/Maze and $\{0\} \cup \mathcal{J}_{\mathrm{fit}}$ for Qwen. Here $\tau_{\mathrm{fit}} > 0$ is the soft-min (fit) temperature, λ_{mean}, λ_{div}, and λ_{res} are nonnegative weights for mean-fit, diversity, and residual penalties, respectively. The implemented normalized soft-min objective is

$$\mathcal{L}_{\mathrm{fit}} = -\frac{\tau_{\mathrm{fit}}}{B} \sum_{b=1}^{B} \log \left[\frac{1}{|\mathcal{T}_b|} \sum_{a \in \mathcal{T}_b} \exp(-\ell_{b,a}/\tau_{\mathrm{fit}}) \right] + \lambda_{\mathrm{mean}} \frac{1}{B} \sum_{b=1}^{B} \frac{1}{|\mathcal{T}_b|} \sum_{a \in \mathcal{T}_b} \ell_{b,a} + \lambda_{\mathrm{div}} \mathcal{L}_{\mathrm{div}} + \lambda_{\mathrm{res}} \mathcal{L}_{\mathrm{res}}. \tag{10}$$

The normalization makes the soft-min term equal to the common branch loss when the branch losses coincide. Exact task-dependent regularizers are given in Appendix C. Inference verification uses no target labels or training-only annotations during held-out evaluation.

Task-specific adapters, native executors, verifiers, and realized-state replanning rules instantiate this shared contract without introducing separate optimization algorithms. The fit losses are task-specific and use labels only inside training: Sudoku uses masked digit cross-entropy at teacher-forced capture states, Maze uses weighted segmentation cross-entropy plus Dice, wall, and endpoint terms, and Qwen uses teacher-forced token cross-entropy on the solution string. Appendix A gives the executable verifier keys, Qwen re-encoding template, and a cross-task interface summary.

4 Experiments

We evaluate TTR on Sudoku, Maze, and MATH500. We first compare complete systems on each task, then use matched controls to examine candidate generation, output selection, and feedback across the full revision process under each task's evaluation budget.

4.1 Experimental Setting

Models and training. We compare TTR with Transformer, TTC, and TTT-Linear (Sun et al., 2024). TTR keeps each task's native model and output head fixed and trains the proposal layer and task adapter. For MATH500, the existing Qwen LoRA adapter also remains fixed. All models are evaluated from frozen checkpoints. TTT retains its within-sequence update during inference.

For Sudoku, the executor is a Transformer (Vaswani et al., 2017) and the proposal layer has rank 64 and radius $\rho = 0.5$. We train it for 1,200 updates on 2,048 boards. Maze uses a spatial executor, rank 32, radius $\rho = 1$, and two groups of four branches. The groups are trained in two stages of 600 updates each on 1,024 size-9 mazes: the first uses initial states, and the second starts from a copy of the first group's parameters and is fitted on states captured at revision steps 0, 1, and 2, with the first group fixed. MATH500 uses Qwen3.5-0.8B+LoRA with rank 8 and radius $\rho = 0.1$. Generation uses bfloat16, a 512-token output limit, temperature $\tau_{\text{gen}} = 0.6$, and nucleus threshold $p_{\text{top}} = 0.95$, without prompt truncation. Appendix B.1 gives the full training configurations and checkpoint identities.

Evaluation and comparisons. The main results use held-out data, with checkpoints, budgets, and selectors fixed before evaluation. Reference labels are used for scoring, not candidate selection. Each baseline retains its own checkpoint and selection rule. Component ablations hold the native executor fixed. The Maze selector replay uses fixed candidate pools from training confirmation slices. We report checkpoint or generation-seed means with sample standard deviations. Paired 95% confidence intervals resample example IDs jointly across checkpoints or generation seeds, with Maze resampling separately within each maze size.

To keep the comparisons interpretable, matched controls hold the stated executor, selector, and generation budget fixed. Candidate-generation controls preserve the executor and verifier, while selector replays apply alternative rules to the same stored candidate pools. State-handoff controls keep the first-round results and round-level budget fixed. Later candidate pools may change when the selected state or prompt changes. Each comparison identifies the component it controls and the resulting downstream changes.

4.2 Tasks and Metrics

Each task uses the same candidate–execution–selection interface, with state updates adapted to its native representation and replanning procedure.

Sudoku. We evaluate the first 1,024 boards of the official held-out hard-Sudoku split (Palm et al., 2018), using the same complete-board decoder for every model. TTR performs one revision with eight branches and the incumbent. This setting evaluates candidate generation and selection in a single round. Board exact accuracy and constraint validity are computed separately. They coincide for all reported main-table rows. Cell accuracy is a secondary metric. Transformer, TTT-Linear, and TTR each use three checkpoints, while TTC uses one fixed checkpoint as the reference throughout.

Maze. We use 256 mazes at each held-out size 11, 13, 15, and 17 from Schwarzschild et al. (2021). TTR runs two revisions, each with eight branches and the incumbent, for 18 candidate executions per maze. The selected hidden map and decoded path are carried into the next revision. Strict branch-free path accuracy is the primary metric. We also report valid-path rate, F1, IoU, wall violations, and endpoint coverage. Metrics are averaged equally over sizes within each checkpoint and then over three checkpoints. TTC uses one checkpoint. Main-table F1 and IoU use pooled pixel counts within each checkpoint–size pair. The paired feedback analysis in Table 9 uses per-example scores.

MATH500. We evaluate all 500 normalized questions from the MATH benchmark (Hendrycks et al., 2021) with ten candidate generations per question and generation seeds 20260815–20260817. TTR uses four branches and the incumbent in each of two rounds. The selected draft is included in the next prompt, which Qwen re-encodes to construct the next state. The coverage-v4 selector uses modal selection in round one and evidence grouping over the cumulative candidate pool in round two. Baselines use modal selection with the same ten-generation budget and answer parser. Exact match

measures the selected answer's accuracy. Oracle@10 measures whether any generated candidate is correct.

Metric names are task-specific. Sudoku validity refers to satisfying all board constraints, and it coincides with complete-board accuracy for the reported rows. Maze Valid measures a connected wall-free path with endpoint coverage, while Strict additionally requires the branch-free path conditions. MATH500 Exact match scores the selected answer, whereas Oracle@10 measures candidate coverage before selection in the reported evaluation on held-out examples.

4.3 Main Results

We first report complete-system results, then use matched comparisons to separate candidate generation, selection, and state handoff. The handoff is adapted to each task: Sudoku uses one revision, Maze carries the selected hidden map and decoded path, and MATH500 re-encodes the selected draft with the question before the next prompt is formed.

Sudoku. Table 1 shows that TTR reaches 96.289% board accuracy and 98.890% cell accuracy. Board accuracy exceeds Transformer by 8.105 percentage points and TTC by 5.469 points. Across the three checkpoints, board accuracy ranges from 96.094% to 96.484%.

Table 1: Sudoku results on 1,024 held-out boards. Values are mean $\pm$ sample SD across three checkpoints. TTC uses one checkpoint. The best value in each metric column is bold.

Model	Board exact/valid	Cell accuracy
Transformer	$88.184 \pm 3.059\%$	$96.696 \pm 0.597\%$
TTC	90.820%	97.323%
TTT-Linear	$37.923 \pm 1.157\%$	$81.135 \pm 0.958\%$
TTR	$\mathbf{96.289 \pm 0.195\%}$	$\mathbf{98.890 \pm 0.050\%}$

Maze. TTR achieves 19.889% Strict accuracy and the highest F1 and IoU among the systems in Table 2. TTC has a higher Valid rate at 91.797%, while its Strict score is zero. TTR reaches 77.995% Valid, showing the distinction between path connectivity and branch-free correctness. Appendix Table 18 further reports zero wall violations and full endpoint coverage for TTR.

Table 2: Maze results on held-out sizes 11/13/15/17, averaged equally across sizes and reported as mean $\pm$ sample SD across three checkpoints. Scores are percentages. TTR uses two revisions. TTC uses one checkpoint. A dash denotes a measured 0% score, with zero mean and SD for multi-checkpoint rows. The best value in each metric column is bold.

Model	Strict	Valid	F1	IoU
Transformer	–	–	52.845 ± 1.499	35.996 ± 1.357
TTC	–	$\mathbf{91.797}$	60.643	43.540
TTT-Linear	–	–	51.631 ± 1.777	34.861 ± 1.626
TTR	$\mathbf{19.889 \pm 0.795}$	77.995 ± 26.262	$\mathbf{71.257 \pm 5.585}$	$\mathbf{56.785 \pm 5.197}$

MATH500. Table 3 reports 37.4% exact match and 51.9% Oracle@10 for TTR under the ten-generation budget. Exact match improves over Transformer, TTC, and TTT-Linear by 11.8, 14.4, and 13.7 percentage points, respectively. The matched component comparison below shows how proposal generation changes candidate coverage and selected-answer accuracy with the Qwen executor fixed.

4.4 Ablation Studies

We next hold the native executors fixed and compare candidate generation, candidate selection, and state handoff. Table 4 summarizes the main comparisons.

Table 3: MATH500 results with ten candidate generations per question. Values are mean ± sample SD across three generation seeds on the same 500 questions. Oracle@10 measures candidate coverage. The best value in each metric column is bold.

Model	Exact match	Oracle@10
Transformer	$25.6 \pm 0.7\%$	$36.6 \pm 0.7\%$
TTC	$23.0 \pm 0.6\%$	$37.4 \pm 0.5\%$
TTT-Linear	$23.7 \pm 0.6\%$	$36.9 \pm 0.7\%$
TTR	$\mathbf{37.4 \pm 1.0\%}$	$\mathbf{51.9 \pm 1.4\%}$

Table 4: Selected component ablations. Scores are percentages, and gains are TTR minus the corresponding control in percentage points. Paired entries follow the metric order in each panel. Maze† uses a fixed-pool replay on training confirmation data. The Sudoku incumbent-only row uses one execution. TTR uses nine.

Ablation	Control	**TTR**	Gain
Sudoku: Board exact			
Incumbent only	92.285	**96.289**	+4.004
Duplicate output	92.285	**96.289**	+4.004
Maze: Strict / Valid			
Random latent	0.977 / 30.143	**19.889 / 77.995**	+18.913 / +47.852
Duplicate output	0.488 / 29.883	**19.889 / 77.995**	+19.401 / +48.112
Stale initial	18.424 / 67.155	**19.889 / 77.995**	+1.465 / +10.840
Confidence only†	15.039 / 28.516	**18.652 / 77.344**	+3.613 / +48.828
MATH500: Exact / Oracle@10			
Stale initial	29.467 / 48.467	**37.400 / 51.867**	+7.933 / +3.400
Draft-feedback rerank	37.267 / 44.533	**37.400 / 51.867**	+0.133 / +7.333

Candidate generation. On Sudoku, candidate proposals and selection raise board accuracy from 92.285% for the incumbent to 96.289%. Repeating the incumbent nine times leaves accuracy at 92.285%. On Maze, learned proposals reach 19.889% Strict accuracy, compared with 0.977% for norm-matched random directions and 0.488% for duplicate outputs at the same 18-execution budget.

With the Qwen checkpoint, LoRA adapter, parser, sampler, and selector fixed, TTR raises Oracle@10 from 44.533% with draft-feedback reranking to 51.867%, a paired gain of 7.333 percentage points with a 95% CI of $[4.733, 10.133]$. Exact match is 37.400% versus 37.267%, a paired difference of $+0.133$ points with a 95% CI of $[-2.600, 2.867]$. The direct comparison separates the gain in candidate coverage from the near-matched final-answer accuracy.

Candidate selection. Using the same Maze candidate pools, structural verification raises Strict accuracy from 15.039% to 18.652% and Valid from 28.516% to 77.344%. The candidate set is unchanged. Only the selection rule differs. Appendix C.6 gives the fixed-pool replay details and the training confirmation slices used for this comparison.

State updates across revisions. State handoff follows each task's native representation. Maze carries the selected hidden map and decoded path, MATH500 re-encodes the selected draft, and Sudoku uses a single revision. On Maze, continuing from the committed state raises Strict by 1.465 percentage points and Valid by 10.840 points over restoring the initial state. The corresponding 95% CIs are $[0.977, 1.986]$ and $[9.831, 11.850]$. A latent-only handoff reaches 20.020% Strict and 67.318% Valid. Explicit path feedback raises Valid to 77.995%.

For MATH500, using the selected-draft prompt for round two instead of restoring the original prompt raises exact match by 7.933 points and Oracle@10 by 3.400 points, with respective 95% CIs of

$[5.133, 10.800]$ and $[0.933, 6.000]$. Figure 3 and Appendix Tables 9 and 13 give the full feedback comparisons.

Candidate budgets. The Maze budget study varies $J \in \{0, 1, 2, 4, 8\}$ at two rounds with early stopping disabled. Branches are taken in group order: $J \leq 4$ uses the first group, while $J = 8$ uses both. Appendix C.5 reports this curve. Timing and peak memory are reported in Appendix C.7.

Taken together, the matched studies identify the role of each module. Proposal learning determines the candidate directions, verification determines which realized candidate is retained, and replanning determines what information reaches the next round. The same accounting applies across the three tasks even though their native state representations differ.

Cross-task interpretation. The three task instantiations expose different parts of the same revision contract. Sudoku places most of the burden on candidate generation and constraint-aware selection because the experiment has one revision round. The complete-board score therefore measures whether a proposed token-state change can produce a globally consistent solution, rather than only improving local cell predictions. The incumbent and duplicate-output controls establish the reference point for interpreting this complete-board comparison under the one-round budget.

Maze makes the state transition explicit because a revision changes both a spatial hidden map and a decoded path. Candidate generation supplies distinct directions, structural verification orders their realized paths, and replanning starts the next round from the selected record. This division is visible in the metrics: learned proposals increase Strict accuracy over norm-matched random directions, structural verification raises Valid and Strict on a fixed pool, and committed-state continuation improves the next execution over restoring the initial map. The latent-only and explicit-path variants further show that the hidden map and realized path carry different information into the next round.

MATH500 uses the same candidate bookkeeping with a language-native state transition. Each selected draft is placed back into the original question context and re-encoded by Qwen before the next round. The draft-feedback reranking control keeps this text-level continuation while changing candidate generation. TTR increases Oracle@10 by 7.333 points, showing that its proposals cover more correct candidates. Exact match remains close between the two systems, which separates coverage from the final selection decision. This separation is useful because it identifies where additional execution helps in the language setting: it broadens the pool presented to the selector while the prompt and parser retain their native semantics throughout revision and answer scoring.

5 Conclusion

TTR is an execution-aware state–proposal–execute–commit–replan contract: bounded latent interventions are provisional, every branch is realized by the native task system, and a label-free key hard-commits one aligned latent/output/realized-state tuple. That tuple, rather than a scalar score or soft mixture, is passed to replanning so that each output remains paired with its corresponding latent state. On frozen evaluations TTR improves complete Sudoku boards and strict Maze paths. More directly, the matched Maze pair shows that committed-state reuse changes subsequent discrete outcomes, while latent-only feedback separates the effects of the latent state and the explicit realized path. On MATH500, feedback expands candidate coverage while selected-answer accuracy remains close to the draft-feedback baseline. TTR therefore provides a reusable execution-grounded transition whose atomic handoff keeps each selected output aligned with its latent and realized state through later revisions under the shared control loop and native execution.

The task adapters determine how the committed record becomes a usable next state without changing the shared proposal and commit logic. This separation lets Sudoku re-encode a board, Maze continue from a selected map and path, and MATH500 rebuild a prompt around a selected draft. The matched controls therefore evaluate one transition contract under three native representations. Together with the fixed-budget audits, these results show that TTR is useful when the executor can expose task-level evidence and the adapter can carry that evidence into the next round.

These results also clarify the role of execution in test-time computation. A candidate is useful when its realized task outcome can be evaluated and its associated state can be handed to the next revision. TTR exposes this interface while leaving task-specific decoding and verification in place.

AI USE STATEMENT

Generative AI tools were used to assist with conceptual framing, methodology and experiment-design feedback, software implementation and review, literature organization, manuscript drafting and editing, and the interpretation and presentation of experimental results. The authors reviewed AI-assisted code and validated it with unit and task smoke tests, checked technical statements against the implementation, experiment artifacts, and cited literature, reviewed all AI-assisted text, and take responsibility for the final content, claims, and artifacts.

REFERENCES

Brandon Amos, Ivan Dario Jimenez Rodriguez, Jacob Sacks, Byron Boots, and J. Zico Kolter. Differentiable MPC for end-to-end planning and control. In *Advances in Neural Information Processing Systems*, 2018. URL https://arxiv.org/abs/1810.13400.

Shaojie Bai, J. Zico Kolter, and Vladlen Koltun. Deep equilibrium models. In *Advances in Neural Information Processing Systems*, 2019. URL https://arxiv.org/abs/1909.01377.

Andrea Banino, Jan Balaguer, and Charles Blundell. PonderNet: Learning to ponder. *arXiv preprint arXiv:2107.05407*, 2021. URL https://arxiv.org/abs/2107.05407.

Douglas A. Bristow, Marina Tharayil, and Andrew G. Alleyne. A survey of iterative learning control. *IEEE Control Systems Magazine*, 26(3):96–114, 2006. doi: 10.1109/MCS.2006.1636313.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021. URL https://arxiv.org/abs/2110.14168.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/abs/1807.03819.

Yilun Du, Jiayuan Mao, and Joshua B. Tenenbaum. Learning iterative reasoning through energy diffusion. In *Proceedings of the 41st International Conference on Machine Learning*, 2024. URL https://arxiv.org/abs/2406.11179.

Alexi Gladstone, Ganesh Nanduru, Md Mofijul Islam, Peixuan Han, Hyeonjeong Ha, Aman Chadha, Yilun Du, Heng Ji, Jundong Li, and Tariq Iqbal. Energy-based transformers are scalable learners and thinkers. In *International Conference on Learning Representations*, 2026. URL https://arxiv.org/abs/2507.02092.

Karol Gregor and Yann LeCun. Learning fast approximations of sparse coding. In *Proceedings of the 27th International Conference on Machine Learning*, 2010. URL https://icml.cc/Conferences/2010/papers/449.pdf.

Etash Guha, Ryan Marten, Sedrick Keh, Negin Raoof, Georgios Smyrnis, Hritik Bansal, et al. OpenThoughts: Data recipes for reasoning models, 2025. URL https://arxiv.org/abs/2506.04178.

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

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring mathematical problem solving with the MATH dataset. *arXiv preprint arXiv:2103.03874*, 2021. URL https://arxiv.org/abs/2103.03874.

Kenji Kubo, Shunsuke Kamiya, Masanori Koyama, Kohei Hayashi, Yusuke Iwasawa, and Yutaka Matsuo. C-Voting: Confidence-based test-time voting without explicit energy functions. In *International Conference on Learning Representations*, 2026. URL https://openreview.net/forum?id=kYQFfEKtx5.

Hunter Lightman, Vineet Kosaraju, Yura Burda, Harri Edwards, Bowen Baker, Teddy Lee, Jan Leike, John Schulman, Ilya Sutskever, and Karl Cobbe. Let's verify step by step. *arXiv preprint arXiv:2305.20050*, 2023. URL https://arxiv.org/abs/2305.20050.

Hongyuan Adam Lu, Z. L. Victor Wei, Qun Zhang, Jinrui Zeng, Bowen Cao, Lingwei Meng, et al. Looped world models. *arXiv preprint arXiv:2606.18208*, 2026. URL https://arxiv.org/abs/2606.18208.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2303.17651.

Rasmus Berg Palm, Ulrich Paquet, and Ole Winther. Recurrent relational networks. In *Advances in Neural Information Processing Systems*, 2018. URL https://arxiv.org/abs/1711.08028.

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

Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588: 604–609, 2020. doi: 10.1038/s41586-020-03051-4. URL https://arxiv.org/abs/1911.08265.

Avi Schwarzschild, Eitan Borgnia, Arjun Gupta, Arpit Bansal, Zeyad Emam, Furong Huang, Micah Goldblum, and Tom Goldstein. Datasets for studying generalization from easy to hard examples, 2021. URL https://arxiv.org/abs/2108.06011.

Noah Shinn, Federico Cassano, Edward Berman, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023. URL https://arxiv.org/abs/2303.11366.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017. URL https://arxiv.org/abs/1712.01815.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, Tatsunori Hashimoto, and Carlos Guestrin. Learning to (learn at test time): RNNs with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 2024. URL https://arxiv.org/abs/2407.04620.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, 2017. URL https://arxiv.org/abs/1706.03762.

Guan Wang, Jin Li, Yuhao Sun, Xing Chen, Changling Liu, Yue Wu, Meng Lu, Sen Song, and Yasin Abbasi Yadkori. Hierarchical reasoning model. *arXiv preprint arXiv:2506.21734*, 2025. URL https://arxiv.org/abs/2506.21734.

Peihao Wang, Shan Yang, Xijun Wang, Tesi Xiao, Xin Liu, Changlong Yu, Yu Lou, Pan Li, Zhangyang Wang, Ming Lin, and Rene Vidal. Beyond test-time training: Learning to reason via hardware-efficient optimal control. *arXiv preprint arXiv:2603.09221*, 2026. URL https://arxiv.org/abs/2603.09221v1.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022. URL https://arxiv.org/abs/2201.11903.

Yudong W. Xu, Wenhao Li, Scott Sanner, and Elias B. Khalil. Self-supervised transformers as iterative solution improvers for constraint satisfaction. In *Proceedings of the 42nd International Conference on Machine Learning*, 2025. URL https://arxiv.org/abs/2502.15794.

Yudong W. Xu, Wenhao Li, Scott Sanner, and Elias B. Khalil. Large neighborhood search meets iterative neural constraint heuristics. *arXiv preprint arXiv:2603.20801*, 2026. URL https://arxiv.org/abs/2603.20801.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022. URL https://arxiv.org/abs/2210.03629.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models. *arXiv preprint arXiv:2305.10601*, 2023. URL https://arxiv.org/abs/2305.10601.

A Task Interfaces

Each task instantiates the state $S_t = (h_t, M_t, C_t, r_t)$ and the four maps in Equation 3. The keys below use the pool index $i \in I_t$ and the smaller-is-better convention of Equation 8. The batch index is suppressed.

These interfaces are related to established learned-iteration, control, fixed-point, and neural-constraint perspectives (Gregor & LeCun, 2010, Amos et al., 2018, Bristow et al., 2006, Bai et al., 2019, Xu et al., 2026). Code paths below are relative to the root of the minimal reproduction repository.

Sudoku. The frozen model produces token hidden states. The adapter combines them with the current board, pooled context, and row, column, and box conflict features. The mutable mask fixes clue cells. Each candidate passes through the frozen head and the confidence-ordered complete decoder. The verifier key is

$$k_{t,i}^{\text{Sudoku}} = \big(n_{\text{clue}}, n_{\text{empty}}, n_{\text{row+column+box}}, -\eta_{\text{Sudoku}}\big). \tag{11}$$

All terms are evaluated on the complete output $\bar{y}_{t,i}$. The three counts measure clue violations, unresolved cells, and constraint conflicts. Validity requires all three to be zero. The confidence aggregate η_{Sudoku} breaks structural ties. The adapter supports re-encoding a selected board for another revision. The reported Sudoku experiments use one round.

The decoder is implemented in `ttr_unified/tasks/sudoku.py`. At each step it masks digits already present in the cell's row, column, or 3×3 box and fills the legal cell–digit pair with the highest native probability. It returns a partial board with reason `no_legal_candidate` if no legal pair remains, or `step_limit` if the decoding budget is exhausted.

Maze. The latent is a channel-last spatial hidden map. The adapter combines it with walkability, endpoints, path neighborhoods, reachability, corridor, and degree features. The segmentation head decodes each candidate into a path raster. The verifier key is

$$\begin{aligned} k_{t,i}^{\text{Maze}} = \big(&\mathbb{1}_{\neg\text{strict}}, \mathbb{1}_{\neg\text{valid}}, n_{\text{wall}}, n_{\text{missing}}, \mathbb{1}_{\text{endpoint-disconnected}}, \\ &\max(n_{\text{components}} - 1, 0), n_{\text{degree}}, n_{\text{raster}}, n_{\text{path}}, -\eta_{\text{Maze}}\big). \end{aligned} \tag{12}$$

The first two entries indicate failure of Strict and Valid, respectively. Valid requires a connected, wall-free raster path. The structural terms are computed on the logical maze grid: n_{wall} counts predicted wall cells, n_{missing} counts uncovered endpoints, and $n_{\text{components}}$ counts path components. The endpoint indicator records disconnection, while n_{degree} sums deviations from degree one at endpoints and degree two at other path nodes. The raster term counts inconsistent 2×2 blocks and predicted border pixels, n_{path} counts logical path cells. Strict requires two covered, connected endpoints, one path component, and zero wall, degree, and raster violations. The confidence aggregate η_{Maze} breaks ties after these terms. Replanning carries the selected hidden map and decoded path into the next state. The stale control restores the initial state before round two.

Qwen. For the frozen Qwen model (Qwen Team, 2026), h_t is the prompt embedding sequence and padding positions are immutable. Candidates use native `inputs_embeds` generation. The MATH500 configuration uses two rounds of five candidates and a cumulative archive of ten outputs. The coverage-v4 selector uses explicit-answer modal selection in round one and ranks answer groups by cross-round agreement, group size, evidence quality, and mean log probability in round two. Within the winning group, evidence quality, likelihood, later-round preference, and earliest index determine the selected draft. Mean log probability provides the fallback when no complete explicit answer is available.

The selected draft is inserted into the next prompt with the original question and a review instruction. Qwen re-encodes this prompt to construct h_{t+1} and C_{t+1} and rebuilds M_{t+1}. The cumulative archive remains available for output selection. Appendix B.1 specifies the prompt template and selector. This selector uses text evidence and agreement. Reference answers are used for offline exact-match scoring.

Table 5 summarizes the three task adaptations.

Table 5: Task adaptations of the shared TTR interface. Sudoku supports board re-encoding. Its reported evaluation uses one revision.

Task	Latent h_t	Native execution	State update
Sudoku	Cell tokens	Head and complete decoder	Board re-encoding
Maze	Spatial hidden map	Segmentation head	Selected map and path
MATH500	Prompt embeddings	Qwen generation	Draft re-encoding

Integration and commit. A task supplies `features`, `execute`, `verify`, and `replan`. The shared layer supplies candidate ordering, incumbent preservation, hard commit, and loop control. Following Equation 9, the execution trace is

$$S_t \to H_t \to E_t \to \mathcal{B}_t \to \mathcal{P}_t \to V_t \to \kappa_t \to S_{t+1}. \tag{13}$$

The execution results E_t are paired with their source latents to form $\mathcal{B}_t$. Each record retains $(h_{t,a}, o_{t,a}, y_{t,a}, \widetilde{r}_{t,a})$, and κ_t is the selected record passed to the task's state update. Sudoku and Maze use structural verifier keys. MATH500 uses the evidence-based selector above.

B Reproducibility and Data Roles

Reproducibility statement. Sections 3 and 4 specify the shared layer, state contract, training objective, and evaluation protocols. Appendix A documents task interfaces. The minimal snapshot contains the unified layer, three task adapters, smoke configurations, scripts, and regression tests. This appendix records data roles, implementation checks, ablations, compute measurements, checkpoint hashes, artifact locations, and per-seed manifests. Pretrained weights, full evaluation datasets, and archived headline results are external inputs identified by their hashes and run manifests.

B.1 Evaluation configurations

Table 6 summarizes the task configurations from the saved YAML files and run manifests. Each task has its own executor, adapter, and proposal parameters.

Fit losses. Sudoku uses digit cross-entropy over four captured teacher-forced states, with clue cells masked and the incumbent excluded from the fit candidate set. The diversity and residual weights are $\lambda_{\text{div}} = 0.05$ and $\lambda_{\text{res}} = 0.002$, with weight decay 10^{-4}. Maze uses weighted pixel cross-entropy with path weight 2, plus Dice, wall, and endpoint terms with weights 0.5, 0.25, and 0.25. Its regularization weights are $\lambda_{\text{mean}} = 0.05$, $\lambda_{\text{div}} = 0.02$, and $\lambda_{\text{res}} = 0.002$, with cosine margin 0.3 and weight decay 10^{-4}.

Qwen uses teacher-forced token cross-entropy on solution strings, $\tau_{\text{fit}} = 0.2$, and $\lambda_{\text{div}} = \lambda_{\text{res}} = 0.01$. Training uses a constant learning rate, branch-major batches of four, and activation checkpointing for

Table 6: Configurations used for the main results. J counts learned branches per round, K counts revision rounds, and ρ bounds each residual coordinate.

Setting	Sudoku	Maze	MATH500
Native model	Transformer S-TR-primary, 6.38M	Frozen Maze checkpoint, 0.194M	Qwen3.5-0.8B with LoRA
Proposal	Rank interaction, post-norm	Spatial layer, padded-topology adapter	TTR layer and task adapter, 4.224M trainable
Rank d_{rank}	64	32	8
J / K / ρ	8 / 1 / 0.5	8 / 2 / 1	4 / 2 / 0.1
Fit data	2,048 boards	1,024 size-9 mazes	4,000 solutions
Fit schedule	1,200 Adam updates	Two stages of 600 Adam updates	Three epochs
Learning rate	3×10^{-3}	3×10^{-3}, then 10^{-3}	2.5×10^{-5}
τ_{fit}	0.35	0.25	0.2
Evaluation data	1,024 held-out boards	256 per size: 11, 13, 15, 17	All 500 questions
Evaluation batch	16	4	8

prompts of at least 384 tokens. As in Equation 10, its fit set $\mathcal{T}_b = \{0\} \cup \mathcal{J}_{\text{fit}}$ includes the incumbent in the five-candidate soft-min normalizer. Gradients update the four learned branches and task adapter. The native backbone and LoRA remain frozen.

B.2 MATH500 GENERATION AND DRAFT FEEDBACK

Table 7 specifies the ten-generation budget. Native generation uses bfloat16, $\tau_{\text{gen}} = 0.6$, $p_{\text{top}} = 0.95$, and a 512-token output limit. Formal prompts are encoded in full without a chat template. The generation temperature τ_{gen} and nucleus threshold p_{top} control sampling, τ_{fit} controls the training soft-min.

Table 7: MATH500 candidate generation in each round. Sampled candidates use $\tau_{\text{gen}} = 0.6$ and $p_{\text{top}} = 0.95$.

Round	Incumbent $a = 0$	Branches $a = 1, \ldots, 4$	Generations
1	Greedy, original prompt	Sampled, original prompt	5
2	Sampled, selected-draft prompt	Sampled, selected-draft prompt	5

The Greedy column in Table 12 reports the first-round incumbent, which is included in this budget. Generation seeds are determined by the registered seed, example index, revision, and branch. The three registered seeds are 20260815–20260817, and all matched controls use the same seed schedule. A zero residual preserves the input latent while native sampling determines the generated text.

Checkpoint and data. The frozen LoRA adapter and data files are identified below. They are external inputs to the minimal package rather than files shipped in its directory. The base-model and TTR checkpoint hashes are recorded in each generation manifest. The fit source is an OpenThoughts-114k slice (Guha et al., 2025).

Frozen LoRA adapter
External frozen adapter, identified by its recorded SHA256
SHA256:
`455435d1907b0cc1a70747c3edce0dd2417b0e134961b2feb115c69d5c22530a`

Fit source: 12,000-row OpenThoughts-114k slice
External OpenThoughts source slice, identified by its recorded SHA256

SHA256:
f496a4456a349393b92cd8d71d80db242a10911955d5b55397b505e85e44082e

Formal evaluation: MATH500
External MATH500 evaluation file, identified by its recorded SHA256
SHA256:
f74e5182c9b44c7cff99245cc4f68276a9a7e4ee1b4a8700154a55985471e835

Fitting uses offset 0 and limit 4,000, development uses offset 4,000 and limit 256. Formal evaluation uses all 500 MATH500 questions. Training caps prompts and targets at 512 and 256 tokens, respectively.

Prompt and state update. The base `math500_boxed` prompt asks for a step-by-step solution ending with "Therefore, the final answer is: [boxed ANSWER]". Round two appends the selected draft using `concise_review_recompute_v4`:

```
<ORIGINAL_PROMPT>

Review the draft answer below.  Check every
mathematical step, correct any error, and solve the
original problem again.  Return a complete answer
ending with the required boxed final answer.  Do not
merely continue the draft.

Draft answer:
<SELECTED_DRAFT>
```

The original prompt and selected draft are inserted verbatim. Sampled branches enable native sampling and use the tokenizer's end-of-sequence token. Both rounds run without early stopping or strict-improvement filtering. The stale control restores the original prompt embeddings and task context for round two. It retains the same cumulative archive, selector, and seed schedule.

Evidence selector. Round one uses the explicit-modal rule based on parseable answers, repeated answer groups, confidence, incumbent presence, and earliest index. Round two uses the cumulative candidate pool $\mathcal{P}_t$. For its record $z_{t,i}$, let $\sigma^{\mathrm{ev}}_{t,i}$ be the evidence strength: 4 for a final tag, 3 for a boxed answer, 2 for an answer marker, and 0 for parser fallback. The quality score is

$$Q^{\mathrm{ev}}_{t,i} = 10\sigma^{\mathrm{ev}}_{t,i} + 4\mathbb{1}_{\mathrm{complete}} - 12\mathbb{1}_{\neg\mathrm{complete}} + 2\mathbb{1}_{\mathrm{late}} - 3\mathbb{1}_{\mathrm{trailing}}. \tag{14}$$

The late indicator marks boxed or final-tag answers in the last 256 characters, while the trailing indicator marks trailing discourse in the normalized answer. Complete candidates with positive evidence strength are grouped by presentation-normalized answer or the frozen mathematical equivalence rule. Groups are ranked lexicographically by cross-round agreement, repeated-answer status, round count, group size, maximum and mean quality, later-round count, mean log probability, and earliest index. Within the winning group, quality, log probability, later-round preference, and earliest index determine the selected record. If no group is eligible, maximum mean log probability with earliest-index ties selects the output. Negating the maximized key coordinates expresses these preferences in the smaller-key convention of Equation 8.

The implementation uses `extract_evidence` and the evidence selector. The relevant source files are:

Evidence selector: `ttr_unified/tasks/llm_qwen_math_evidence_v3.py`
Equivalence routine: `ttr_unified/tasks/_legacy.py`
Fast wrapper: `ttr_unified/tasks/llm_qwen_math_fast_v4.py`

B.3 Implementation and evaluation records

The package is named `ttr-unified-layer-try` and imports as `ttr_unified`. The shared `TTRRevisionLayer` is implemented in `ttr_unified/layer.py`. The files `state.py` and `interfaces.py` define the state and task contract. Task adapters are in `ttr_unified/`

`tasks/`, while `experiment.py`, the scripts in `scripts/`, and the configurations in `configs/` coordinate training, evaluation, checkpoints, and manifests. The recorded validation includes compile checks, the package's CPU unit tests, and native smoke tests covering masks, incumbent preservation, lexicographic ties, gradients, task interfaces, commit, checkpoint loading, resumption, and proposal groups.

Fit labels enter the training loss, while inference uses task inputs and candidate outputs. Formal main results and official-test controls use the held-out examples listed in Table 6. The fixed-pool Maze selector replay and additional structured diagnostics use training confirmation slices: Sudoku indices 2560–3583 after development selection on 3584–4095, and Maze indices 512–767. Selector replay uses stored candidate pools from these confirmation slices. Controls evaluated after checkpoint selection are recorded as frozen evaluations. Paired intervals resample example IDs jointly across checkpoints or generation seeds, with Maze sampling separately within each size.

Each archive records the data role, checkpoint hashes, predictions, per-seed metrics, execution counts, device, dtype, elapsed time, and peak memory. Baseline evaluations use the same sample IDs and native evaluators as TTR. The Maze budget package contains 84 completed leaves and 233,472 candidate executions, the reviewer-control supplement contains 33 leaves, including Sudoku execution controls and Maze direction and feedback controls. The artifact paths are listed in Appendix C.10.

C Additional Implementation Detail

C.1 Proposal parameterizations

Let $\mathcal{N}_g$ denote the configured normalization, $\mathcal{G}$ the shared dense gate, P_w the fixed projection retaining the first w feature channels ($1 \le w \le D$), and $\mathcal{U}_g$ the learned up-projection from d_{rank} to D channels. Concatenation along the channel axis is written $[\,,\,]$. The implemented gates are

$$v_t^g = \begin{cases} \tanh(\mathcal{G}([\mathcal{N}_g(h_t), f_t])), & \text{dense gate,} \\ \mathcal{N}_g([h_t, P_w f_t]), & \text{direct gate,} \end{cases} \qquad q_t^g = \mathcal{D}_g(v_t^g). \tag{15}$$

Here v_t^g is the gated proposal input and q_t^g is the rank-space feature. For learned branch embedding e_j^g and rank transform Ψ_g, the branch direction in Equation 4 is

$$\widehat{\Delta h}_{t,j}^{\,g} = \begin{cases} \tanh\big(\mathcal{U}_g(q_t^g) + e_j^g\big), & \text{hidden-additive,} \\ \tanh\big(\mathcal{U}_g(\Psi_g(q_t^g + e_j^g))\big), & \text{rank-interaction.} \end{cases} \tag{16}$$

The embedding e_j^g lies in $\mathbb{R}^D$ for hidden-additive conditioning and $\mathbb{R}^{d_{\text{rank}}}$ for rank-interaction, broadcast over batch and positions. Here ξ is a rank-space input and $\Psi_g(\xi) = \tanh(\xi)$ by default. The spatial Maze variant instead uses

$$\Psi_g(\xi) = \mathrm{SiLU}\big(\mathrm{Conv}_{1\times1}^g\big(\mathrm{SiLU}\big(\xi + \mathrm{DWConv}_{3\times3}^g(\xi)\big)\big)\big). \tag{17}$$

Here $\mathrm{Conv}_{1\times1}^g$ and $\mathrm{DWConv}_{3\times3}^g$ are pointwise and depthwise spatial convolutions, respectively, $\tanh$ and SiLU are standard pointwise activations. In Equation 5, γ_j^g denotes the stored scale after clamping to $[0, 1]$, or one if scaling is disabled.

C.2 Training regularizers

For sample b and learned candidate $a \in \mathcal{J}_{\text{fit}}$, let $\Delta h_{b,a}$ be its residual, with the training state index suppressed and branches enumerated across groups as in Section 3. Set $d_{b,a} = \mathrm{vec}(\Delta h_{b,a})$ and $\bar{d}_{b,a} = d_{b,a} / \max(\|d_{b,a}\|_2, \epsilon)$. Here vec flattens all latent coordinates, $\|\cdot\|_2$ is the Euclidean norm, and $\epsilon = 10^{-12}$ for Sudoku/Qwen and 10^{-6} for Maze. Let $J_{\text{fit}} = |\mathcal{J}_{\text{fit}}|$. For $a, a' \in \mathcal{J}_{\text{fit}}$, set $\zeta_{b,aa'} = \bar{d}_{b,a}^{\mathsf{T}} \bar{d}_{b,a'}$ and $R_b = (\zeta_{b,aa'})_{a,a'} \in \mathbb{R}^{J_{\text{fit}} \times J_{\text{fit}}}$, T denotes transpose and $\mathrm{triu}(R_b, 1)$ retains the strict upper triangle. The implemented regularizers are

$$\begin{aligned} \mathcal{L}_{\text{res}} &= \mathrm{Mean}_{b,a,\nu,\chi}[(\Delta h_{b,a,\nu,\chi})^2], \\ \mathcal{L}_{\text{div}} &= \begin{cases} \frac{1}{B J_{\text{fit}}^2} \sum_{b=1}^{B} \|\,\mathrm{triu}(R_b, 1)\|_F^2, & \text{Sudoku/Qwen,} \\ \mathrm{Mean}_{b,a<a'}[\max(0, \zeta_{b,aa'} - 0.3)^2], & \text{Maze.} \end{cases} \end{aligned} \tag{18}$$

Here Mean is the arithmetic mean over the displayed indices, including immutable coordinates for $\mathcal{L}_{\mathrm{res}}$, ν and χ index position and channel. The squared Frobenius norm $\|\cdot\|_F^2$ sums squared matrix entries. Sudoku/Qwen divide by BJ_{fit}^2, including the zeroed entries, Maze averages only strict pairs, with cosine margin 0.3. For $J_{\mathrm{fit}} < 2$, both diversity penalties are zero. The incumbent is excluded from both regularizers. Maze trains the two four-branch groups in successive stages of 600 updates each. The first group is fitted on initial states with learning rate 3×10^{-3}. Its parameters initialize the second group, which is fitted with learning rate 10^{-3} on states captured at revision steps 0, 1, and 2 using the first group. The first group and shared parameters remain fixed during this stage. Evaluation loads both groups from the resulting checkpoint ($J = 8$).

C.3 Candidate modes

The code exposes four modes through the same task interface: These rows correspond to

Table 8: Candidate modes used in the component comparisons.

Mode	Candidate latent	Purpose
Learned	learned bounded residuals	full layer
Random	norm-matched random signs	direction control
Draft-feedback rerank	identical latents	execution/selection control
Incumbent only	branch zero only	frozen executor reference

`internal_cclr`, `random_latent`, `output_rerank_only`, and `incumbent_only`, respectively. The corresponding evaluations use the official-test protocols or fixed confirmation candidate pools listed below.

C.4 Official-test Maze feedback ablation

The feedback pair uses the main Maze examples, three checkpoints, two rounds, and nine candidates per round. Only the second-round state changes. Results average the four sizes within each checkpoint and report mean ± sample SD. For this paired analysis, each maze e contributes $\mathrm{F1}_e = 2\mathrm{TP}_e/(2\mathrm{TP}_e + \mathrm{FP}_e + \mathrm{FN}_e)$ and $\mathrm{IoU}_e = \mathrm{TP}_e/(\mathrm{TP}_e + \mathrm{FP}_e + \mathrm{FN}_e)$, where TP, FP, and FN count path pixels. These scores are averaged equally over examples, sizes, and checkpoints. Table 2 instead computes the ratios from pooled pixel counts within each checkpoint–size pair.

Table 9: Realized-state feedback versus stale-state reuse on Maze. Scores are percentages, differences and intervals are percentage points. All scores use per-example averaging. The 95% intervals use 10,000 paired bootstrap resamples of example IDs within each size, jointly across checkpoints.

Metric	Realized committed	Stale initial	δ_{pair}	95% CI
Strict	$\mathbf{19.889 \pm 0.795}$	18.424 ± 0.881	$+1.465$	$[0.977, 1.986]$
Valid	$\mathbf{77.995 \pm 26.262}$	67.155 ± 25.082	$+10.840$	$[9.831, 11.850]$
F1	$\mathbf{74.771 \pm 6.732}$	74.742 ± 6.730	$+0.029$	$[-0.042, 0.096]$
IoU	$\mathbf{65.364 \pm 5.505}$	65.262 ± 5.445	$+0.102$	$[0.013, 0.186]$

Realized feedback improves Strict by 1.465 points and Valid by 10.840 points over stale-state reuse. Mean per-example F1 and IoU increase by 0.029 and 0.102 points, respectively.

C.5 Official-test budget and matched-control audit

The official-test Maze package evaluates $J \in \{0, 1, 2, 4, 8\}$ and matched controls at $J = 8$. Each leaf contains 256 examples and two revision rounds. Values are checkpoint means after equal-weight size macro-averaging.

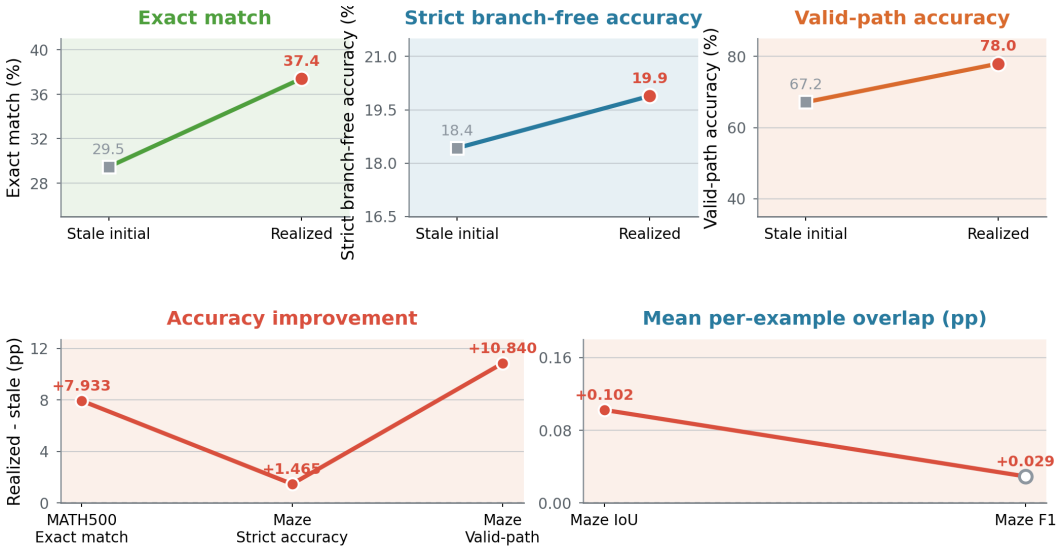


Figure 3: Absolute and paired results for the realized-feedback comparisons on Maze and MATH500. Maze F1 and IoU use mean per-example scores as in Table 9. The paired F1 interval includes zero.

Table 10: Official-test Maze budget curve. Scores are percentages. Candidate executions are totals over the 12 checkpoint–size leaves, and wall time is seconds per checkpoint. Best scores and lowest costs are bold.

J	Strict	Valid	F1	IoU
0	0.488 ± 0.000	29.883 ± 0.000	39.668 ± 0.000	26.396 ± 0.000
1	17.318 ± 1.430	31.576 ± 0.226	40.640 ± 0.117	27.317 ± 0.109
2	17.318 ± 1.430	41.634 ± 16.550	52.059 ± 19.802	38.286 ± 19.013
4	17.383 ± 1.378	41.634 ± 16.550	52.142 ± 19.859	38.382 ± 19.107
8	$\mathbf{19.889 \pm 0.795}$	$\mathbf{77.995 \pm 26.262}$	$\mathbf{71.257 \pm 5.585}$	$\mathbf{56.785 \pm 5.197}$

J	Candidate executions	Wall time (s)
0	**6,144**	$\mathbf{91.5 \pm 4.2}$
1	12,288	92.6 ± 3.2
2	18,432	115.3 ± 20.2
4	30,720	126.1 ± 15.9
8	55,296	230.4 ± 57.3

Table 11: Official-test Maze matched controls at $J = 8$. Scores are percentages, all rows use 55,296 candidate executions. Wall time is seconds per checkpoint. Best scores and lowest wall time are bold.

Mode	Strict	Valid	F1	IoU
Latent-only feedback	$\mathbf{20.020 \pm 0.775}$	67.318 ± 25.029	$\mathbf{71.277 \pm 5.477}$	$\mathbf{56.812 \pm 5.057}$
Random latent	0.977 ± 0.098	30.143 ± 0.056	39.453 ± 0.019	26.254 ± 0.015
Output-only duplicate	0.488 ± 0.000	29.883 ± 0.000	39.668 ± 0.000	26.396 ± 0.000
Stale initial	18.424 ± 0.881	67.155 ± 25.082	71.235 ± 5.582	56.755 ± 5.189
TTR realized	19.889 ± 0.795	$\mathbf{77.995 \pm 26.262}$	71.257 ± 5.585	56.785 ± 5.197

Mode	Wall time (s)
Latent-only feedback	268.9 ± 24.9
Random latent	$\mathbf{129.7 \pm 5.0}$
Output-only duplicate	133.0 ± 3.5
Stale initial	224.8 ± 14.2
TTR realized	230.4 ± 57.3

At the matched budget, realized feedback gives 19.889% Strict and 77.995% Valid. The stale-state comparison gives paired gains of $+1.465$ and $+10.840$ points, respectively. The remaining rows provide candidate direction and duplicate-execution references.

C.6 Component and confirmation controls

The component package contains the execution, selection, and feedback controls used in the main text. Its source is:

```
ttr_unified/
tests/
```

The minimal snapshot contains the control implementations and regression tests, while the stored candidate pools used for the reported aggregates remain external result inputs.

Table 12: MATH500 component comparison under the same ten-generation budget. Oracle@10 measures candidate coverage. Scores are percentages.

MATH500 mode	Exact match	Greedy	Oracle@10
Draft-feedback rerank (no latent perturbation)	37.267 ± 0.702	$\mathbf{17.200}$	44.533 ± 0.577
Random latent	36.667 ± 1.007	$\mathbf{17.200}$	46.200 ± 0.529
Learned, stale initial	29.467 ± 0.462	$\mathbf{17.200}$	48.467 ± 1.102
TTR, realized feedback	$\mathbf{37.400 \pm 1.000}$	$\mathbf{17.200}$	$\mathbf{51.867 \pm 1.419}$

Here "Greedy" is the first native generation, and "Oracle@10" is the fraction of questions for which any of the ten generated answers matches the reference under the frozen offline parser. The MATH500 rows use 500 questions per generation seed and three seeds. Paired intervals in Table 13 resample question IDs jointly across all three seeds.

Selection from a fixed candidate pool. An offline replay uses the same 18 stored candidates in each pool. Structural verification gives 18.652% Strict and 77.344% Valid, compared with 15.039% and 28.516% for confidence-only selection. The source is:

Table 13: Question-level paired differences for the frozen MATH500 controls. Intervals use 10,000 resamples of question IDs jointly across the three generation seeds. Values are percentage points.

Comparison	Metric	δ	95% CI
TTR − stale	Exact match	$+7.933$	$[5.133, 10.800]$
TTR − stale	Oracle@10	$+3.400$	$[0.933, 6.000]$
TTR − draft feedback	Exact match	$+0.133$	$[-2.600, 2.867]$
TTR − draft feedback	Oracle@10	$+7.333$	$[4.733, 10.133]$

Table 14: Official-test Sudoku execution controls with the same frozen native checkpoint and one revision. Incumbent-only uses one candidate, the duplicate-output control uses nine. Scores are percentages.

Mode	Board exact	Valid	Cell accuracy
Incumbent only	$\mathbf{92.285 \pm 0.000}$	$\mathbf{92.285 \pm 0.000}$	$\mathbf{97.596 \pm 0.000}$
Output-only duplicate	$\mathbf{92.285 \pm 0.000}$	$\mathbf{92.285 \pm 0.000}$	$\mathbf{97.596 \pm 0.000}$

Table 15: Maze confirmation-slice feedback comparison. Scores are percentages.

Maze confirmation mode	Strict	Valid path	F1
Stale initial	16.829 ± 0.407	66.992 ± 25.541	70.749 ± 5.589
TTR, realized feedback	$\mathbf{18.652 \pm 0.775}$	$\mathbf{77.344 \pm 27.491}$	$\mathbf{70.775 \pm 5.605}$

`ttr_unified/layer.py`
`tests/test_hard_commit.py`

The archived replay pool is not part of the minimal code snapshot. The listed files provide the deterministic hard-commit implementation and its regression coverage.

Table 16: Official-test Maze TTR results by checkpoint and logical size. Scores are percentages for 256 examples per row. F1 uses pooled pixel counts. Best scores within each size are bold. The large Valid SD in the aggregate is concentrated in the checkpoint–size cells, while Strict is comparatively stable.

Size	Checkpoint	Strict	Valid	F1
11	Seed 42	33.2	80.1	**85.4**
11	Seed 43	35.5	85.9	83.3
11	Seed 44	**36.3**	**98.4**	83.8
13	Seed 42	19.9	63.7	**79.0**
13	Seed 43	19.9	83.2	77.8
13	Seed 44	**20.7**	**100.0**	75.7
15	Seed 42	18.4	34.4	58.6
15	Seed 43	16.4	85.9	**72.6**
15	Seed 44	**18.8**	**99.6**	69.7
17	Seed 42	6.6	16.8	36.5
17	Seed 43	5.5	87.9	**67.6**
17	Seed 44	**7.4**	**100.0**	65.0

The checkpoint-size table gives the per-leaf Maze values underlying the aggregate results.

C.7 INFERENCE COST AUDIT

The audit reports elapsed time and peak memory per leaf on one RTX 4090 D. Times include model loading, candidate execution, verification, and aggregation. Training and dataset download are excluded.

Table 17: Inference time, peak memory, and parameter counts for structured tasks. The lowest cost in each column is bold within each task.

Task	Model	Time (s)	Peak MiB	Parameters (M)
Sudoku	TTC	104.1	211.9	6.65
Sudoku	Transformer	**33.0**	**48.7**	**6.38**
Sudoku	TTT-Linear	853.9	51.1	6.67
Sudoku	**TTR**	75.3	128.4	6.48
Maze	TTC	5.2	66.7	0.220
Maze	Transformer	**1.2**	25.5	0.418
Maze	TTT-Linear	36.9	**23.3**	0.428
Maze	**TTR**	62.0	135.8	**0.194**

For MATH500, TTR times for seeds 20260815/16/17 are 6031.3/8304.8/8053.7 s. Matched TTC times are 11625.6/11297.8/11075.8 s. These times include tokenization, re-encoding, generation, and selection.

C.8 MAZE DIAGNOSTIC ENDPOINTS

The supporting Maze diagnostics use the same aggregation as Table 2:

Table 18: Formal Maze wall-violation and endpoint-coverage diagnostics.

Model	Wall violation	Endpoint coverage
Transformer	$6.721 \pm 2.554\%$	$71.745 \pm 7.156\%$
TTC	$\mathbf{0.000}\%$	$\mathbf{100.000}\%$
TTT-Linear	$9.947 \pm 2.284\%$	$72.465 \pm 7.506\%$
TTR	$\mathbf{0.000 \pm 0.000}\%$	$\mathbf{100.000 \pm 0.000}\%$

C.9 HARD-COMMIT INVARIANT

For a verifier key tensor of shape $[B, N, L_k]$, where B is the batch size, N is the active per-round or cumulative pool size, and L_k is the number of lexicographic key coordinates, TTR initializes each batch item at pool index zero. It scans candidates in ascending index order and locates the first unequal key coordinate. A replacement occurs only when that coordinate is smaller than the current best coordinate. This implementation provides deterministic first-index selection without depending on floating point score aggregation. In a per-round pool, index zero is the unchanged incumbent. In the cumulative MATH500 pool, it is the earliest retained candidate.

C.10 RESULT ARTIFACTS

The minimal reproduction package contains the following relevant directories:

Unified layer and task adapters
`ttr_unified/`

Training and evaluation entry points
`scripts/`

Smoke configurations
`configs/`

Regression tests
`tests/`

Local smoke outputs
`results/`

The last directory is created by local runs and is not pre-populated in the code snapshot. Pretrained weights, full evaluation data, and archived headline results are kept outside this minimal package so that the snapshot remains small and its smoke checks remain portable.