# VERIFIED COGNITIVE SCHEMA TRANSFER FOR TEST-TIME LEARNING

**Anonymous authors**
Paper under double-blind review

## ABSTRACT

Test-time scaling improves the problem-solving capabilities of large language models (LLMs) by allocating additional inference compute to each problem. Existing memory-augmented methods enable problem-solving experience to accumulate and transfer across tasks, but accumulated experience does not automatically yield a systematic understanding of solution principles. A key challenge is to use verification feedback to abstract structures shared across concrete problem-solving experiences and organize them into cognitive schemas with explicit applicability conditions that guide subsequent reasoning. To address this challenge, we introduce Verified Cognitive Schema Transfer (VCST), a framework for accumulating and reusing knowledge during inference without updating model parameters. VCST extracts task-level schemas from verified solutions, abstracts shared solution structures across tasks, and reuses the resulting schemas after checking their applicability conditions. For new tasks, it retrieves and applies relevant knowledge through tag-based filtering and semantic reranking. Under a fixed API budget, VCST adaptively balances cognition construction and task solving. Experiments across four benchmark suites, 13 benchmarks, and nine backbone LLMs show that VCST improves accuracy over strong test-time scaling baselines. On LiveCodeBench, VCST improves the final solve rate over S$^*$ by 6.1–20.7 percentage points across nine backbone models, and enables a 7B model to outperform a 32B model equipped with S$^*$. VCST also generalizes broadly, improving over the strongest corresponding baselines by 2.3–2.4 points on multi-task reasoning and an average of 7.5 points across the six CRUXEval settings. On multi-language code generation, VCST achieves the highest accuracy in all six evaluated languages.

## 1 INTRODUCTION

Large language models have shown strong performance on reasoning and coding tasks. A growing body of work on *test-time scaling* further improves performance by allocating additional inference compute, for instance through parallel sampling, search, and iterative refinement. Representative systems such as S$^*$ (Li et al., 2025) combine these techniques to deliver substantial gains on competitive programming benchmarks.

Per-instance test-time scaling primarily allocates computation to solving the current problem. Memory-augmented methods such as ExpeL (Zhao et al., 2024), A-MEM (Xu et al., 2025), and ReasoningBank (Ouyang et al., 2026) extend this process by retaining and reusing experience across tasks. However, accumulated experience does not automatically yield a systematic understanding of solution principles. A key challenge is to use verification feedback to identify shared solution structures and organize them into cognitive schemas with explicit applicability conditions. We argue that inference should both solve the current task and build such schemas to guide subsequent reasoning, allowing experience gained on earlier tasks to inform later problem solving.

To this end, we propose **Verified Cognitive Schema Transfer** (VCST), which augments a pretrained LLM with a persistent task-stream state and a verified schema memory built from execution-validated experience. VCST extracts schemas from verifier-passing solutions, abstracts shared solution structures across tasks, and reuses them after checking their applicability. For new tasks, it retrieves relevant knowledge through tag-based filtering, condition checking, and semantic reranking, while adaptively balancing knowledge acquisition and task solving under a fixed API budget.



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.

We evaluate VCST across four benchmark suites. On LiveCodeBench, VCST achieves the highest final solve rate on all nine backbone models, outperforming S* by 6.1–20.7 percentage points. Qwen-Coder-7B with VCST achieves a higher solve rate than Qwen-Coder-32B with S* (72.6% vs. 68.3%). Beyond competitive programming, VCST improves over the best-performing baseline, ReasoningBank, by 2.3–2.4 percentage points on multi-task reasoning. Across the four code-state prediction benchmarks, VCST achieves the highest accuracy for all three backbone models, with further gains over native extended-reasoning modes where available. This advantage extends to multi-language code generation, where VCST improves over the strongest baseline in each of the six languages by 6.12–13.33 percentage points.

**Contributions.**

- We propose VCST, a test-time scaling framework that turns execution-validated experience into reusable local cognition, converting inference compute from a per-problem cost into a cumulative investment.
- We couple cross-task schema abstraction with applicability-aware retrieval, distilling shared solution structures from verified solutions and checking their applicability before reuse on new tasks.
- We validate VCST across four benchmark suites (13 benchmarks in total), including nine backbones on LiveCodeBench, showing accuracy gains over strong test-time scaling baselines.

## 2 Related Work

### 2.1 Test-Time Scaling

Test-time scaling improves LLM performance by allocating additional inference compute to each problem. Self-Consistency (Wang et al., 2023) samples diverse reasoning chains and aggregates them by majority voting. Tree of Thoughts (Yao et al., 2023) performs deliberate search over partial solutions, while Self-Refine (Madaan et al., 2023), Reflexion (Shinn et al., 2023), and CRITIC (Gou et al., 2024) improve answers through iterative critique, revision, and tool feedback. In code generation, S* (Li et al., 2025) combines parallel sampling, execution-grounded debugging, and adaptive selection, forming a strong single-problem baseline.

Recent work also studies how inference compute should be allocated. Snell et al. (2025) analyze the tradeoff between parallel breadth and sequential revision depth. TALE (Han et al., 2025) incorporates token-budget awareness into generation, s1 (Muennighoff et al., 2025) improves reasoning through budget forcing, and DeepSeek-R1 (Guo et al., 2025) shows that reinforcement learning with verifiable rewards can induce reasoning traces that benefit from additional compute.

These methods substantially improve per-problem performance, but their computation remains largely instance-bound: samples, search traces, critiques, and verified attempts are mainly consumed within the current problem. They do not organize execution-validated experience into persistent cognitive structures that can be reused across related future tasks. VCST addresses this limitation by treating test-time inference as a cumulative process in which solving a task can also construct reusable, verification-grounded knowledge.

### 2.2 Memory-Augmented Agents

A complementary line of work equips agents with persistent memory across interactions. Generative Agents (Park et al., 2023) maintain memory streams and distill them into higher-level reflections, while Voyager (Wang et al., 2024) accumulates executable skills for embodied exploration. More recent methods study memory organization and cross-task transfer directly: ExpeL (Zhao et al., 2024) distills natural-language insights from task trajectories without parameter updates; A-MEM (Xu et al., 2025) creates linked, evolving memory notes; and ReasoningBank (Ouyang et al., 2026) distills self-judged successful and failed experiences into reusable reasoning strategies. These methods establish experience retention as a useful axis for test-time improvement, but their memory units are primarily recalled or organized from prior interactions.

VCST targets the same persistent task-stream setting while making the transfer criterion explicit: a schema is extracted from an execution-verified trajectory, checked against its applicability scope, and conditionally injected into a subsequent task. Thus, the comparison is not whether an agent stores

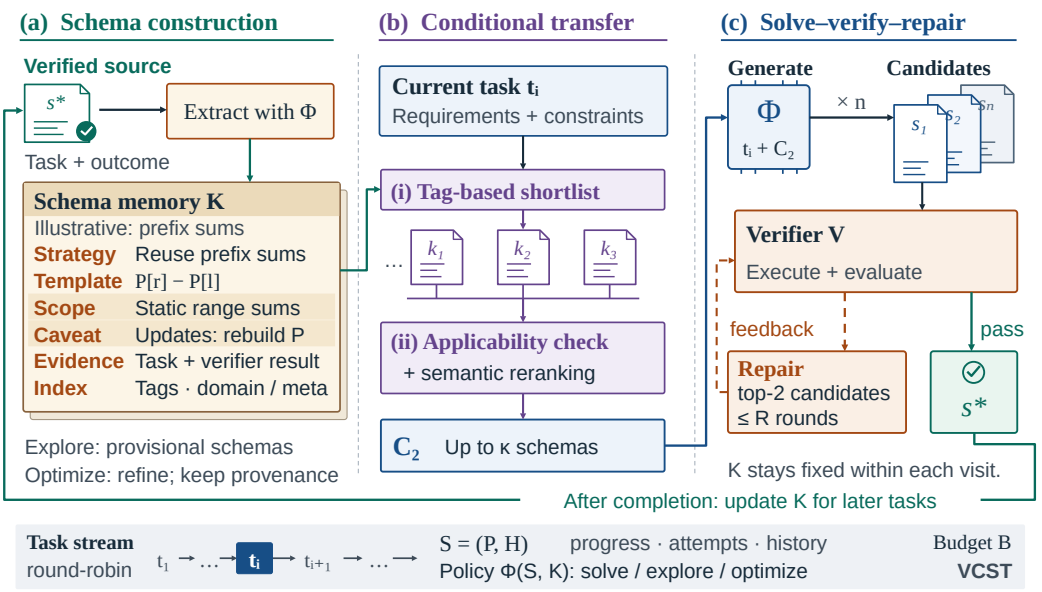



Figure 1: Overview of VCST. (a) Verified solutions populate schema memory $K$. (b) Tag filtering, applicability checking, and semantic reranking retrieve schemas to guide (c) candidate generation and verification-guided repair. Task-stream state $S$ supports round-robin scheduling under budget $B$. The prefix-sum schema is illustrative; $P$ denotes prefix sums, with half-open ranges $[l, r)$.

history, but whether stored experience is recalled directly or transferred as a condition-scoped schema. ExpeL, A-MEM, and ReasoningBank therefore constitute the most direct baselines for evaluating VCST.

## 3 Methodology

The key insight underlying VCST is that solving a verifiable task generates a byproduct that conventional methods discard: execution-grounded evidence of what works—algorithmic patterns, solution templates, and observed failure signals specific to the current task cluster. VCST systematically captures this evidence as a persistent local cognitive framework and transfers it to structurally related subsequent tasks, turning per-problem compute into a cumulative investment (Figure 1).

Formally, we consider a fixed task batch $T = \{t_1, t_2, \ldots, t_n\}$ presented in a predefined order.

VCST uses a shared verification procedure $V$ to obtain task-specific feedback $v_i(s)$. A candidate solution $s$ is accepted if $\text{verify}(s, t_i, V) = 1$.

The agent operates under a model-call budget $B$, covering solution generation and repair as well as action selection, schema extraction, tagging, applicability checking and reranking, exploration, and optimization whenever these operations invoke an LLM.

### 3.1 Verified Cognitive Schema Transfer

**Definition 1** ( Verified Cognitive Schema Transfer). *A VCST system is a tuple $M = (\Phi, S, K)$, where $\Phi$ denotes a pretrained LLM, $S$ the task-stream state, and $K$ the verified schema memory.*

As shown in Figure 1, $S$ records task progress and history, while $K$ stores reusable schemas that condition later task solutions.

**Task-Stream State.** The task-stream state $S = (P, H)$ tracks task progress $P = \{(t_i, \sigma_i, e_i)\}$, where $\sigma_i \in \{\text{unsolved}, \text{partial}, \text{solved}\}$ and $e_i$ is the attempt count, together with the interaction history $H$ recording generated solutions, verification outcomes, and debugging traces.

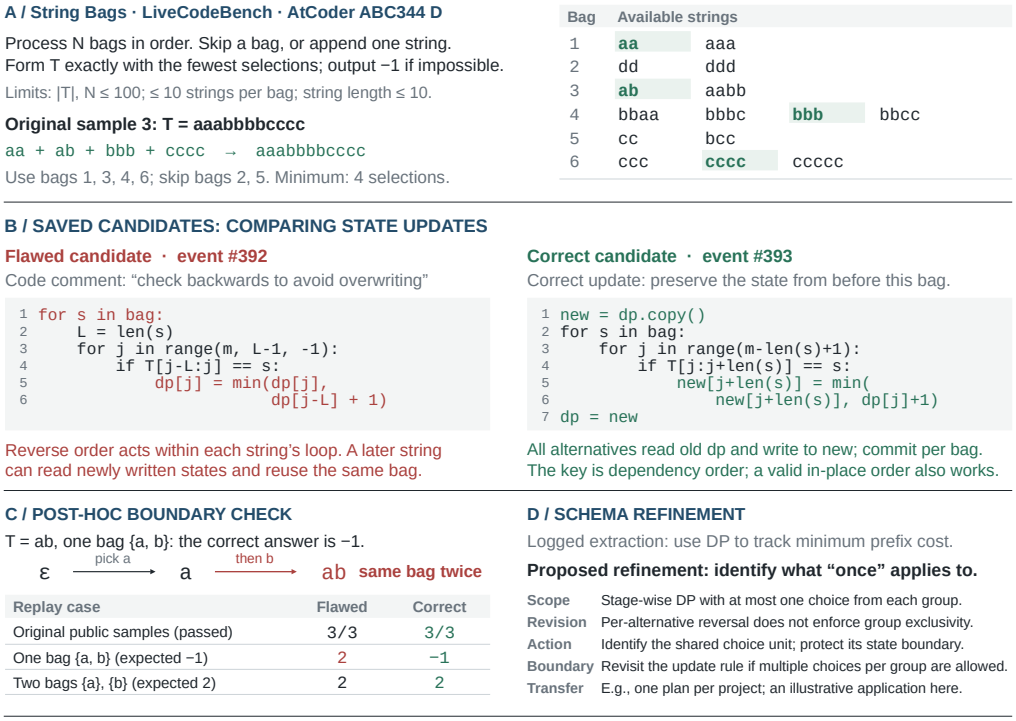



Figure 2: Condition-aware schema refinement illustrated through String Bags (LiveCodeBench; AtCoder ABC344 D). (A) The original task and a public sample. (B) Two saved GPT-4o-mini candidates generated in parallel, with abbreviated code. (C) Post-hoc replay shows that both pass the three public samples, while a constructed one-bag input exposes unintended reuse. (D) An author-proposed refinement adds applicability and state-update boundaries to the logged DP summary; the project-selection mapping is illustrative.

**Verified Schema Memory.** The schema memory $K = \{k_1, \ldots, k_m\}$ stores reusable reasoning patterns extracted from prior experience. Each entry records an insight, applicability conditions and caveats, a solution template, tags, a type label, and provenance linking it to one or more source tasks and their verification outcomes. Figure 2 uses a String Bags candidate analysis to illustrate how task constraints can inform a schema's applicability conditions.

### 3.2 Knowledge Construction and Retrieval

**Verification-First Acquisition.** Knowledge extraction from LLM reasoning alone is susceptible to hallucination: plausible-sounding but incorrect patterns can corrupt $K$ and mislead future retrieval. VCST grounds schema admission in verifier feedback (Figure 1(a)). We adopt a verification-first update rule:

$$K_{i+1} = K_i \cup \text{Extract}(\Phi, t_i, s^*, v_i(s^*)) \quad \text{if} \quad \text{verify}(s^*, t_i, V) = 1, \tag{1}$$

where $s^*$ is the first candidate passing verification and $\text{Extract}(\cdot)$ summarizes it into a structured entry grounded in execution evidence. This gate applies to entries promoted to the verified schema memory. During *explore*, synthesized entries remain provisional until they obtain verifier-linked provenance; only then can they be promoted to $K$ and retrieved. During *optimize*, REFINE may consolidate schemas from multiple tasks by abstracting a shared solution structure and parameterizing task-specific details. The consolidated schema states the conditions required for reuse and retains each source's verification record.

**Two-Stage Retrieval.** Given a new task $t_q$, the two-stage mechanism shown in Figure 1(b) selects the most relevant entries from $K$. First, coarse filtering uses Jaccard similarity over tag sets:

$$\text{score}(k_j, t_q) = \frac{|\,\text{tags}_j \cap \tau(t_q)|}{|\,\text{tags}_j \cup \tau(t_q)|}, \tag{2}$$

where $\tau(t_q) = \Phi(t_q)$ is the tag set generated by $\Phi$ for $t_q$ (defined as 0 when both sets are empty). We discard entries with zero tag overlap, rank the rest by Jaccard similarity, and form a shortlist $C_1$.

In the second stage, $\Phi$ jointly assesses applicability and semantic relevance using the task description and each candidate's insight, applicability conditions, and tags. Here, $\text{Compat}(k_j, t_q) = 1$ denotes an LLM judgment of applicability; up to $\kappa$ schemas are returned as $C_2$, with an empty selection when none is judged applicable.

### 3.3 Adaptive Decision and Execution

At each outer step, $\Phi$ selects an operation type from $\{\text{solve}, \text{explore}, \text{optimize}\}$ using the current task-stream state $S$ and schema memory $K$; a deterministic scheduler selects the task when the operation is *solve*. The action-selection prompt uses a compact summary of $S$ rather than the complete interaction history. The full solver—task scheduling, candidate generation, repair, and verification-first knowledge update—is summarized in Algorithm 1 in Appendix G. Within each task visit (Figure 1(c)), tasks unresolved after $R$ repair rounds remain pending for a later scheduled visit.

## 4 Experiments

### 4.1 Competitive Programming

Competitive programming combines algorithmic diversity with unambiguous test-case verification, making it the natural primary testbed for VCST. We use this setting to test whether knowledge accumulation yields consistent gains over a leading test-time scaling baseline across diverse model families and parameter scales.

**Experimental Setup.** We report solve rate on LiveCodeBench v2 (Jain et al., 2025), comprising 511 problems released between May 2023 and May 2024 on Codeforces, LeetCode, and AtCoder, with 182 Easy, 206 Medium, and 123 Hard problems. We evaluate nine backbone models from three families: the GPT series (GPT-4o-mini, GPT-4.1-nano, GPT-5-nano), Qwen-Coder (Qwen2.5-Coder-Instruct) at 7B, 14B, and 32B, and DeepSeek-R1-Distill at 7B, 14B, and 32B (Guo et al., 2025). We compare against Direct Inference, which uses a single greedy decoding pass, and S$^*$ (Li et al., 2025), which runs 16 parallel samples with up to three self-debugging rounds and represents the current state of the art on this benchmark. We evaluate all methods on the same 511-problem manifest using the test execution and scoring procedures adopted by S$^*$. For Figure 3, VCST generates five candidates per task visit, ranks them using verification feedback, and repairs the top two candidates for up to three rounds. Round-robin scheduling permits revisits to unresolved tasks using the updated schema memory. ExpeL, A-MEM, and ReasoningBank use the same candidate-generation and repair settings, retrieve at most two memory items per visit, and retain their respective memory construction and update mechanisms. Schema memory is initialized empty for each run and remains fixed within each visit. Model-call and token counts include generation, repair, memory operations, and model-call retries; detailed configurations and cost accounting are provided in Appendix F.

**Results.** Figure 3 presents results across all nine backbone models. Our method achieves the highest final solve rate on all nine evaluated backbones. The improvement over S$^*$ ranges from $+6.1$pp to $+20.7$pp, while the improvement over Direct Inference ranges from $+22.7$pp to $+43.2$pp. The magnitude of the gain depends on both the backbone family and model scale: it is largest for the Qwen-Coder models and decreases with scale in the R1-Distill family. We also observe that smaller models augmented with our method surpass larger models running S$^*$: Qwen-Coder-7B (72.6%) outperforms Qwen-Coder-32B with S$^*$ (68.3%), and GPT-4o-mini (76.5%) exceeds GPT-4.1-nano with S$^*$ (72.8%). This cross-scale transfer suggests that accumulated knowledge effectively compensates for model capacity limitations, with direct practical implications for deployment cost.

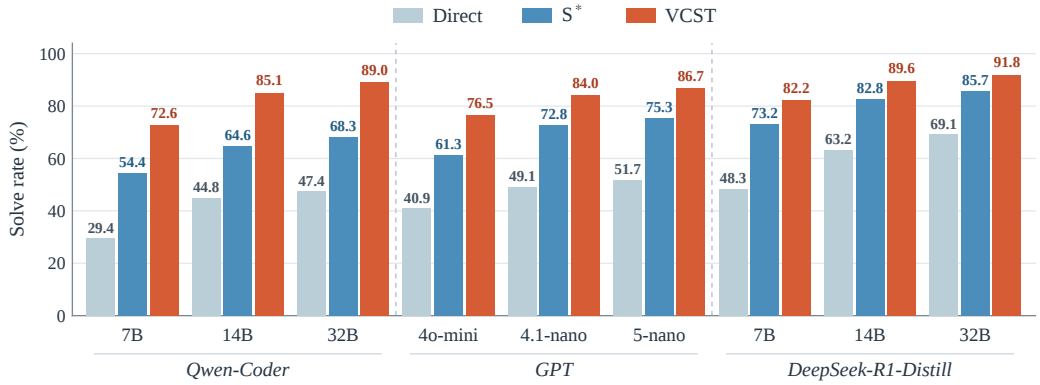



Figure 3: Final solve rates on LiveCodeBench across nine backbone models. We compare Direct generation, S* with parallel sampling and iterative debugging, and VCST with schema memory and deferred revisits.

Table 1: Pass@1 accuracy and average API calls per problem (API/P) of VCST across nine backbone models on LiveCodeBench (511 problems). S* uses at least 16 calls per problem. Bold: best pass@1.

| | GPT Series | | | Qwen-Coder Series | | | R1-Distill Series | | |
|---|---|---|---|---|---|---|---|---|---|
| | 4o-mini | 4.1-nano | 5-nano | 7B | 14B | 32B | 7B | 14B | 32B |
| Pass@1 (%) | 73.0 | 81.6 | 82.2 | 65.6 | 80.4 | 83.2 | 76.5 | 86.1 | **88.3** |
| API/P | 6.1 | 6.9 | 6.3 | 7.0 | 6.5 | 5.8 | 7.9 | 7.2 | 6.4 |

Table 1 further reports pass@1 accuracy and average API calls per problem (API/P) across all nine backbone models. In this evaluation, each task is visited once, with a single round of five-candidate generation and no repair or deferred revisit. Memory updates made after the visit are used only for subsequent tasks. S* allocates a fixed budget of 16 parallel samples per problem, yielding a minimum of 16 API calls before debugging; VCST instead uses $n = 5$ parallel samples and performs no repair in this evaluation, reducing this to 5.8–7.9 calls, a 2–3× reduction. R1-Distill-32B attains the highest pass@1 (88.3%) at 6.4 calls, and Qwen-Coder-32B reaches 83.2% at only 5.8 calls, demonstrating that knowledge-guided generation yields strong first-round solutions without expensive parallel sampling. The full multi-round configuration additionally repairs failed candidates and invokes deferred retry on unsolved problems; this extra budget is spent only where pass@1 fails, preserving average efficiency on the long tail of solved instances.

**Failure Analysis.** Under GPT-4.1-nano, some unresolved problems pass most tests but fail on edge cases, including integer overflow and boundary conditions. VCST and S* solve partially different subsets of problems, suggesting potential complementarity between schema-guided solving and per-problem search.

### 4.2 Multi-Task Reasoning

To assess whether cross-problem knowledge accumulation generalizes beyond code generation, we evaluate on seven benchmarks spanning mathematical reasoning, code generation, and general reasoning, covering domains where verified feedback takes fundamentally different forms.

**Experimental Setup.** We evaluate on MATH (Hendrycks et al., 2021) and GSM8K (Cobbe et al., 2021) for mathematical reasoning, MBPP (Austin et al., 2021) and HumanEval+ (Liu et al., 2023) for code generation, and BBH (Suzgun et al., 2023), LongBench (Bai et al., 2024), and MMLU-CF (Zhao et al., 2025) for reasoning and knowledge; we report benchmark scores on a 0–100 scale and their unweighted mean across the seven benchmarks. Each task is attempted once, and correctness feedback is obtained only after the answer is submitted. This feedback is used solely for schema construction

Table 2: Scores (0–100) on multi-task reasoning benchmarks. The best result in each column is bolded and the second best is underlined.

| | | Math | | Code | | Reasoning & Knowledge | | | |
|---|---|---|---|---|---|---|---|---|---|
| Model | Method | MATH | GSM8K | MBPP | HumanEval+ | BBH | LBench | MMLU-CF | Avg. |
| GPT-4o-mini | CoT | 78.3 | 90.9 | 72.4 | 68.9 | 78.3 | 57.6 | 69.6 | 73.7 |
| | Reflexion | 80.8 | 92.6 | 72.9 | 77.4 | 76.8 | 62.7 | 70.0 | 76.2 |
| | CRITIC | 81.9 | 92.7 | 74.1 | 82.3 | 78.9 | 58.9 | 70.3 | 77.0 |
| | ExpeL | 83.1 | 93.8 | 75.3 | 83.5 | 83.2 | 64.1 | 70.2 | 79.0 |
| | A-MEM | 83.6 | 94.7 | 74.9 | 84.1 | 83.7 | 65.5 | 70.9 | 79.6 |
| | ReasoningBank | 83.3 | 95.1 | 75.6 | 83.5 | 84.2 | 64.7 | 71.5 | 79.7 |
| | **VCST** | **85.2** | **97.3** | **78.6** | **87.2** | **85.8** | **67.2** | **73.6** | **82.1** |
| DeepSeek V3 | CoT | 82.2 | 96.2 | 75.7 | 84.1 | 80.0 | 59.6 | 74.5 | 78.9 |
| | Reflexion | 86.8 | 98.2 | 78.4 | 89.0 | 79.8 | 64.7 | 73.9 | 81.5 |
| | CRITIC | 87.6 | 98.0 | 80.1 | 88.4 | 80.1 | 62.1 | 76.0 | 81.8 |
| | ExpeL | 91.7 | 97.7 | 79.1 | 89.6 | 81.4 | 63.5 | 76.1 | 82.7 |
| | A-MEM | 91.5 | 98.1 | 80.4 | 89.0 | 81.1 | 63.9 | 77.4 | 83.1 |
| | ReasoningBank | 90.6 | 98.4 | 79.6 | 89.0 | 80.8 | **70.8** | 76.1 | 83.6 |
| | **VCST** | **92.8** | **98.6** | **84.2** | **94.5** | **82.5** | 70.6 | **78.2** | **85.9** |

for subsequent tasks. Baselines span three categories: non-iterative prompting via CoT, iterative refinement via Reflexion (Shinn et al., 2023) and CRITIC (Gou et al., 2024), and persistent memory via ExpeL (Zhao et al., 2024), A-MEM (Xu et al., 2025), and ReasoningBank (Ouyang et al., 2026). We use GPT-4o-mini and DeepSeek-V3 as backbones, and re-implement all baselines under a unified protocol.

**Results.** Table 2 shows that VCST achieves the highest average on both backbones, reaching 82.1% on GPT-4o-mini and 85.9% on DeepSeek-V3. This exceeds the strongest memory baseline, ReasoningBank, by $+2.4$pp and $+2.3$pp, respectively. Relative to the strongest baseline in each column, VCST gains $+1.6$/$+1.1$pp on MATH, $+2.2$/$+0.2$pp on GSM8K, $+3.0$/$+3.8$pp on MBPP, and $+3.1$/$+4.9$pp on HumanEval+ for GPT-4o-mini/DeepSeek-V3.

VCST leads on 13 of 14 backbone–benchmark combinations. The only exception is LongBench with DeepSeek-V3, where ReasoningBank leads by $0.2$pp; with GPT-4o-mini, VCST leads LongBench by $1.7$pp. On BBH and MMLU-CF, the gains over the strongest baseline are $+1.6$/$+1.1$pp and $+2.1$/$+0.8$pp, respectively.

### 4.3 Code State Prediction

To validate whether VCST builds genuine predictive capability over program behavior rather than merely surface-level pattern matching, we evaluate on four code reasoning benchmarks requiring explicit program state tracking across complementary dimensions.

**Experimental Setup.** We evaluate on four benchmarks and report accuracy on each: CruxEval-O and CruxEval-I (Gu et al., 2024) test forward and inverse execution prediction on 800 samples each, HaltEval-prelim (HaltEval) (FAIR CodeGen Team, Meta, 2025) covers binary halting classification on 230 samples, and QuixBugs (Lin et al., 2017) tests single-line bug repair on 40 programs; each problem is attempted once; as in the multi-task evaluation, correctness feedback is obtained after answer submission and used only to construct schemas for subsequent tasks. Baselines are Direct inference, CoT (Wei et al., 2022) prompting, and Reasoning mode, which extends self-directed reasoning at temperature 0.7 and is available for Qwen3-32B and DeepSeek-R1-Distill-32B only. We select GPT-4o-mini, Qwen3-32B with optional Reasoning mode, and the reasoning-specialized DeepSeek-R1-Distill-32B to assess whether VCST's benefits persist alongside strong internal reasoning.

**Results.** Table 3 shows consistent improvements across all four benchmarks and three models, with our method achieving the highest accuracy in every setting. On execution prediction benchmarks, accumulated knowledge serves as state patterns for program state tracking: on CruxEval-I (inverse

Table 3: Accuracy (%) on code reasoning benchmarks. R1-32B denotes DeepSeek-R1-Distill-32B; "Reasoning" denotes the model's extended reasoning mode.

| Benchmark | GPT-4o-mini | | | Qwen3-32B | | | | R1-32B | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Direct | CoT | **VCST** | Direct | CoT | Reasoning | **VCST** | Direct | CoT | Reasoning | **VCST** |
| HaltEval | 61.7 | 65.7 | **81.3** | 63.0 | 76.5 | 84.3 | **87.8** | 75.2 | 81.3 | 85.2 | **90.4** |
| CruxEval-O | 63.8 | 83.3 | **87.6** | 67.5 | 74.3 | 86.1 | **92.4** | 81.0 | 85.8 | 89.4 | **92.6** |
| CruxEval-I | 41.5 | 46.0 | **60.8** | 46.3 | 49.6 | 54.3 | **62.1** | 44.5 | 47.8 | 52.6 | **61.3** |
| QuixBugs | 67.5 | 70.0 | **85.0** | 70.0 | 72.5 | 77.5 | **85.0** | 85.0 | 87.5 | 90.0 | **95.0** |

execution prediction), our method outperforms CoT by $+14.8$pp, $+12.5$pp, and $+13.5$pp across the three models; on CruxEval-O (forward prediction), Qwen3-32B improves by $+18.1$pp over CoT and $+6.3$pp over its own Reasoning mode. These gains across both forward and inverse execution prediction suggest that VCST is useful across complementary forms of program-state reasoning, including settings with strong extended-reasoning baselines.

On HaltEval and QuixBugs, VCST consistently outperforms the native Reasoning modes of Qwen3-32B and R1-32B, suggesting that its schema-guided solving framework offers further gains beyond these reasoning baselines. On HaltEval, GPT-4o-mini (81.3%) approaches Qwen3-32B's Reasoning mode (84.3%), while DeepSeek-R1-Distill-32B reaches 90.4%, exceeding its own Reasoning mode by $+5.2$pp; on QuixBugs, GPT-4o-mini improves by $+15.0$pp over CoT, Qwen3-32B surpasses its Reasoning mode by $+7.5$pp, and DeepSeek-R1-Distill-32B reaches 95.0%, a $+5.0$pp gain over its Reasoning mode. One possible explanation is that retrieved schemas help the model relate a new bug to reusable repair strategies, guiding its analysis of the faulty code.

### 4.4 Multi-Language Code Generation

Algorithmic strategies can be reused across languages when related tasks share the same underlying computational structure. We study multi-language code generation with a fixed backbone, comparing VCST with direct generation, test-time search, and memory-augmented baselines.

**Experimental Setup.** We use the full Aider Polyglot Benchmark, covering Python, JavaScript, Java, Go, Rust, and C++, with GPT-5-mini as the common backbone. All methods are evaluated on the same task set using the language-specific test suites. We compare Direct, ExpeL, A-MEM, ReasoningBank, S$^*$ ($N = 8, R = 2$), and VCST. Accuracy is the fraction of tasks whose generated solutions pass the associated tests, with per-language sample counts given in the figure caption. Average API calls per task are measured over the full evaluation set and include solution generation, repair, and all auxiliary model calls, including memory operations.

**Results.** Figure 4(a) shows a consistent advantage for VCST across all six programming languages. Although the strongest competing method varies by language, VCST leads in every case, improving over the best baseline in each language by 6.12–13.33 percentage points. This pattern suggests that the benefits of the framework extend across the evaluated language settings. Figure 4(b) places these gains in the context of inference cost: VCST uses 9.3 API calls per task, compared with 10.5 for A-MEM, 12.8 for ReasoningBank, and 11.4 for S$^*$.

### 4.5 Ablation Study

To quantify the contribution of each component, we conduct ablation experiments on a controlled subset of 240 LiveCodeBench problems (80 Easy, 80 Medium, 80 Hard) using GPT-5-nano. We systematically remove or downgrade components from the full VCST $M = (\Phi, S, K)$. All pairwise comparisons are validated with bootstrap significance tests ($N = 3{,}000$ resamples); differences are significant at $p < 0.05$ unless noted.

**Results.** Table 4 shows the effects of individual ablations under the evaluated configuration. (a) KB: removing $K$ entirely costs $-4.2$pp ($p < 0.001$), concentrated on hard problems (75.0%→63.8%); removing solution templates while retaining insights and tags reduces accuracy by 5.8 percentage

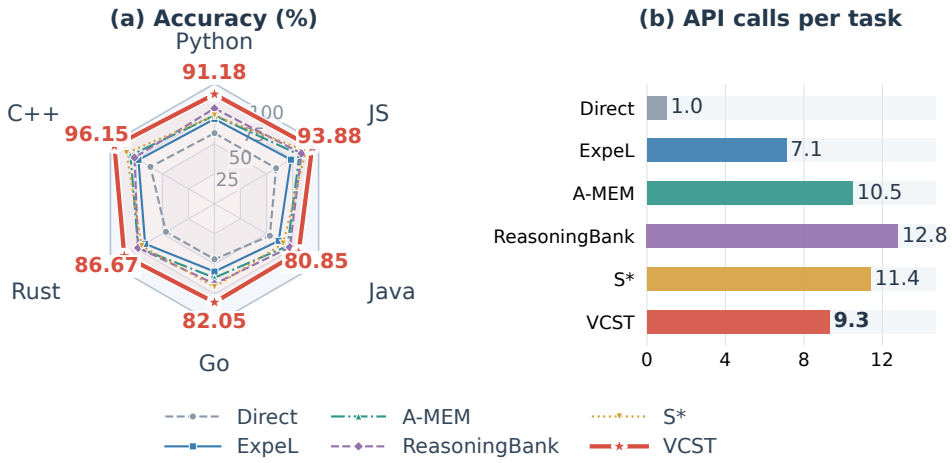



Figure 4: Aider Polyglot results with GPT-5-mini. (a) Accuracy (%) on Python (34 tasks), JavaScript (49), Java (47), Go (39), Rust (30), and C++ (26). (b) Average API calls per task.

Table 4: Ablation study on LiveCodeBench (240 problems, GPT-5-nano). Each group removes or downgrades one component of $M = (\Phi, S, K)$. Bold: best in column.

| Component | Variant | KB ($K$) | Retrieval | Scheduling | Easy | Med. | Hard | Overall |
|---|---|---|---|---|---|---|---|---|
| – | **Full ( VCST)** | ✓ | 2-stage | Round-Robin | **100.0** | **90.0** | **75.0** | **88.3** |
| (a) KB | w/o KB | × | – | Round-Robin | **100.0** | 88.8 | 63.8 | 84.2 |
| | w/o Template* | ✓ | 2-stage | Round-Robin | **100.0** | 80.0 | 67.5 | 82.5 |
| (b) Retrieval | Jaccard Only (1-stage) | ✓ | 1-stage | Round-Robin | **100.0** | **90.0** | 66.3 | 85.4 |
| | Random Retrieval | ✓ | Random | Round-Robin | 98.8 | 83.8 | 60.0 | 80.8 |
| (c) State-Based Opt. | w/o State Summary | ✓ | 2-stage | Round-Robin | **100.0** | 87.5 | 67.5 | 85.0 |
| | Sequential (single pass) | ✓ | 2-stage | Sequential | 98.8 | 85.0 | 66.3 | 83.3 |

*Template code is removed from $K$; insights and tags are retained.

points ($p < 0.001$), suggesting that concrete templates help translate stored insights into usable solutions in this setting. (b) Retrieval: replacing two-stage with Jaccard-only costs $-2.9$pp ($p = 0.004$); random retrieval costs $-7.5$pp ($p < 0.001$), which is larger than the cost of removing $K$ entirely—irrelevant retrieved entries actively harm generation, confirming that precise selection is essential. (c) State-based optimization: The w/o State Summary variant removes the task-progress and interaction-history summary supplied to the LLM, while retaining completion flags and visit counts for multi-pass round-robin scheduling. Accuracy decreases by 3.3 percentage points ($p = 0.002$).

The Sequential (single pass) variant reduces accuracy by 5.0 percentage points. Under this setting, unresolved problems receive no further task visits, limiting opportunities to solve them with subsequently accumulated schemas. Candidate generation and repair within each visit are retained.

## 5 Conclusion

We introduced VCST, which turns verified problem-solving experience into reusable cognitive schemas through cross-task abstraction and applicability-aware retrieval. Across four benchmark suites, including nine LiveCodeBench backbones, VCST improves accuracy over the compared test-time scaling baselines, suggesting that accumulated schemas can complement per-problem inference compute. Two limitations remain: schema memory grows with accumulated experience, and retrieval still relies partly on coarse tags. Future work may explore hierarchical memory organization, learned dense retrieval, and broader verification mechanisms. VCST offers a step toward inference systems that build reusable cognitive structure while solving tasks.

## AI USE STATEMENT

We used generative AI to assist with literature retrieval, code implementation, and interpretation of experimental results. The methodology was proposed by the authors, who take responsibility for the final paper.

## REFERENCES

Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, and Charles Sutton. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, Yuxiao Dong, Jie Tang, and Juanzi Li. LongBench: A bilingual, multitask benchmark for long context understanding. In *Annual Meeting of the Association for Computational Linguistics*, 2024.

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.

FAIR CodeGen Team, Meta. CWM: An open-weights LLM for research on code generation with world models. *arXiv preprint arXiv:2510.02387*, 2025.

Zhibin Gou, Zhihong Shao, Yeyun Gong, Yelong Shen, Yujiu Yang, Nan Duan, and Weizhu Chen. CRITIC: Large language models can self-correct with tool-interactive critiquing. In *International Conference on Learning Representations*, 2024.

Alex Gu, Baptiste Roziere, Hugh Leather, Armando Solar-Lezama, Gabriel Synnaeve, and Sida I. Wang. CRUXEval: A benchmark for code reasoning, understanding and execution. In *International Conference on Machine Learning*, 2024.

Daya Guo, Dejian Yang, Haowei Zhang, et al. DeepSeek-R1 incentivizes reasoning in LLMs through reinforcement learning. *Nature*, 645:633–638, 2025. doi: 10.1038/s41586-025-09422-z.

Tingxu Han, Zhenting Wang, Chunrong Fang, Shiyu Zhao, Shiqing Ma, and Zhenyu Chen. Token-budget-aware LLM reasoning. In *Findings of the Association for Computational Linguistics: ACL*, 2025.

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. In *NeurIPS Datasets and Benchmarks Track*, 2021.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. LiveCodeBench: Holistic and contamination free evaluation of large language models for code. In *International Conference on Learning Representations*, 2025.

Dacheng Li, Shiyi Cao, Chengkun Cao, Xiuyu Li, Shangyin Tan, Kurt Keutzer, Jiarong Xing, Joseph E. Gonzalez, and Ion Stoica. $S^*$: Test time scaling for code generation. In *Findings of the Association for Computational Linguistics: EMNLP 2025*, pp. 15964–15978, 2025. doi: 10.18653/v1/2025.findings-emnlp.865. URL https://aclanthology.org/2025.findings-emnlp.865/.

Derrick Lin, James Koppel, Angela Chen, and Armando Solar-Lezama. QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by ChatGPT really correct? rigorous evaluation of large language models for code generation. In *Advances in Neural Information Processing Systems*, 2023.

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.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori Hashimoto. s1: Simple test-time scaling. In *Conference on Empirical Methods in Natural Language Processing*, 2025.

Siru Ouyang, Jun Yan, I-Hung Hsu, Yanfei Chen, Ke Jiang, Zifeng Wang, Rujun Han, Long T. Le, Samira Daruki, Xiangru Tang, Vishy Tirumalashetty, George Lee, Mahsan Rofouei, Hangfei Lin, Jiawei Han, Chen-Yu Lee, and Tomas Pfister. ReasoningBank: Scaling agent self-evolving with reasoning memory. In *International Conference on Learning Representations*, 2026.

Joon Sung Park, Joseph C. O'Brien, Carrie Jun Cai, Meredith Ringel Morris, Percy Liang, and Michael S. Bernstein. Generative agents: Interactive simulacra of human behavior. In *ACM Symposium on User Interface Software and Technology*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*, 2023.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling LLM test-time compute optimally can be more effective than scaling parameters for reasoning. In *International Conference on Learning Representations*, 2025.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V. Le, Ed H. Chi, Denny Zhou, and Jason Wei. Challenging BIG-Bench tasks and whether chain-of-thought can solve them. In *Findings of the Association for Computational Linguistics: ACL*, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An open-ended embodied agent with large language models. *Transactions on Machine Learning Research*, 2024.

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

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, 2022.

Wujiang Xu, Zujie Liang, Kai Mei, Hang Gao, Juntao Tan, and Yongfeng Zhang. A-MEM: Agentic memory for LLM agents. In *Advances in Neural Information Processing Systems*, 2025.

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. In *Advances in Neural Information Processing Systems*, 2023.

Andrew Zhao, Daniel Huang, Quentin Xu, Matthieu Lin, Yong-Jin Liu, and Gao Huang. ExpeL: LLM agents are experiential learners. In *AAAI Conference on Artificial Intelligence*, 2024.

Qihao Zhao, Yangyu Huang, Tengchao Lv, Lei Cui, Qinzheng Sun, Shaoguang Mao, Xin Zhang, Ying Xin, Qiufeng Yin, Scarlett Li, and Furu Wei. MMLU-CF: A contamination-free multi-task language understanding benchmark. In *Annual Meeting of the Association for Computational Linguistics*, 2025.

## A VERIFICATION-GATE DIAGNOSTIC

To audit the admission rule in isolation, we compare VCST with an otherwise matched control that additionally admits schemas induced from failed trajectories. Table 5 reports the expanded diagnostic on 128 examples under a shared configuration.

Table 5: Verification-gate diagnostic on 128 examples. "Unverified" counts entries induced from failed trajectories.

| Condition | Solved / 128 | Accuracy (%) | Verified | Unverified | Requests | Tokens |
|---|---|---|---|---|---|---|
| Verified-only admission | 94/128 | 73.4 | 92 | 0 | 2,805 | 3.55M |
| Admit failed-trajectory schemas | 88/128 | 68.8 | 90 | 113 | 3,271 | 4.87M |

The verified-only condition solves 94 of 128 examples and retains 92 verifier-linked schemas and no unverified schemas, whereas the admission control additionally retains 113 schemas induced from failed trajectories. Verified-only admission is $6/128 = 4.69$ percentage points higher in accuracy while using 466 fewer requests and 1.32M fewer tokens. These expanded results support the intended provenance boundary and show that unrestricted admission of failed-trajectory schemas can increase memory-maintenance cost without improving aggregate accuracy.

## B MEMORY ABLATION ACROSS BENCHMARKS

Table 6 compares VCST with Matched NoMemory on GPT-4o-mini to examine the contribution of cross-task memory. Both conditions use the same evaluation tasks; Matched NoMemory disables cross-task memory while retaining VCST's other solver settings. MATH, MBPP, BBH, and HumanEval+ follow Section 4.2; CRUXEval-I/O follow Section 4.3. LiveCodeBench uses the generation and repair settings of Section 4.1, reporting first-visit pass@1 on a fixed 120-problem subset.

Table 6: Memory ablation across benchmarks. Accuracy is in %; tokens are per target task. NoMem denotes Matched NoMemory.

| Metric | CRUXEval-I | CRUXEval-O | MBPP | BBH | MATH | HumanEval+ | LiveCodeBench |
|---|---|---|---|---|---|---|---|
| NoMem accuracy | 52.3 | 85.4 | 72.0 | 84.0 | 90.0 | 86.0 | 58.3 |
| VCST accuracy | 60.8 | 87.6 | 78.0 | 86.0 | 86.0 | 87.2 | 73.3 |
| NoMem tokens | 3,493 | 2,523 | 8,857 | 6,377 | 19,759 | 2,831 | 23,661 |
| VCST tokens | 3,851 | 3,049 | 9,456 | 8,902 | 20,781 | 4,306 | 29,818 |

VCST improves accuracy on six of seven benchmarks, with the largest gain on the 120-problem LiveCodeBench subset (58.3% to 73.3%); MATH decreases from 90.0% to 86.0%. Token use increases on every benchmark, reflecting the additional cost of constructing and using memory.

## C TASK-ORDER ROBUSTNESS

We evaluate VCST and its variant without the memory component on the LiveCodeBench task set under three independently shuffled task orders generated with prespecified permutation seeds. Both variants use the same order in each comparison, with VCST's memory initialized empty for each run and all other evaluation settings held fixed. Table 7 reports accuracy and tokens per problem for each order, together with their mean and sample standard deviation across orders.

Table 7: Accuracy and token use across three fixed task orders. The final column in each group reports the mean and sample standard deviation across orders.

| | Accuracy (%) | | | | Tokens/target | | | |
|---|---|---|---|---|---|---|---|---|
| Method | Order 1 | Order 2 | Order 3 | Mean ± SD | Order 1 | Order 2 | Order 3 | Mean ± SD |
| VCST w/o Memory | 57.9 | 58.3 | 58.3 | 58.2 ± 0.2 | 24,075 | 23,850 | 23,059 | 23,661 ± 534 |
| VCST | 76.9 | 77.1 | 76.5 | 76.8 ± 0.3 | 29,979 | 29,767 | 29,707 | 29,818 ± 143 |

Across the three task orders, VCST obtains an average accuracy of 76.8%, compared with 58.2% for its variant without the memory component. Accuracy and token use vary little across the three orders, with standard deviations of 0.3 percentage points and 143 tokens per target for VCST.

A separate five-order VCST evaluation on the same 511-problem manifest breaks down accuracy by difficulty in Table 8. Overall accuracy ranges from 76.1% to 77.1%, with a mean of 76.7% and a sample standard deviation of 0.4 percentage points.

Table 8: Difficulty-stratified VCST accuracy (%) over five task orders. Tasks gives the problem count; Std. dev. is the sample standard deviation in percentage points. These orders are separate from those in Table 7.

| Difficulty | Tasks | Seed 1 | Seed 2 | Seed 3 | Seed 4 | Seed 5 | Mean | Std. dev. |
|---|---|---|---|---|---|---|---|---|
| Easy | 182 | 87.4 | 89.0 | 86.8 | 88.5 | 86.8 | 87.7 | 1.0 |
| Medium | 206 | 76.2 | 76.7 | 76.2 | 76.7 | 75.7 | 76.3 | 0.4 |
| Hard | 123 | 61.0 | 60.2 | 62.6 | 60.2 | 61.0 | 61.0 | 1.0 |
| Overall | 511 | 76.5 | 77.1 | 76.7 | 76.9 | 76.1 | 76.7 | 0.4 |

## D SCHEMA-MEMORY GROWTH OVER THE TASK STREAM

To characterize how memory evolves during batch co-solving, Figure 5 traces a GPT-4.1-nano run on the 511-problem LiveCodeBench manifest. The schema memory grows throughout the run, while the cumulative coverage of both Medium and Hard problems continues to increase. Hard problems account for 14% of the first 50 solved problems and 54% of problems 401–450 in the chronological solve order. The later part of the solution stream therefore contains a substantially larger share of Hard problems alongside the accumulated schema memory.

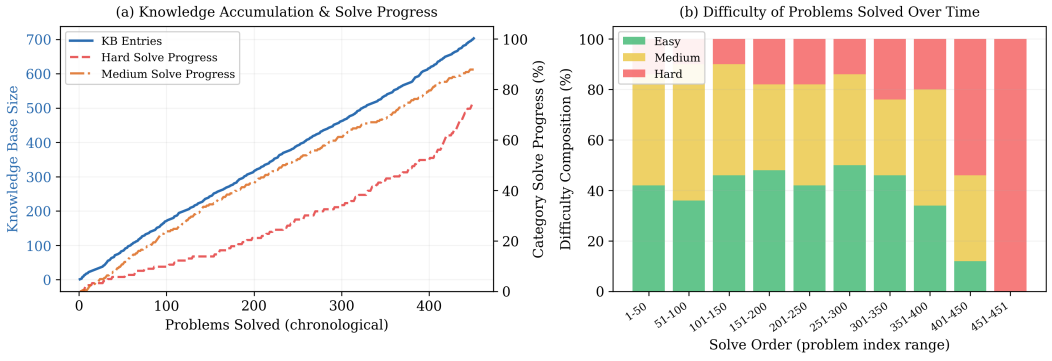



Figure 5: Schema-memory accumulation and solve progression in a GPT-4.1-nano run on the 511-problem LiveCodeBench manifest. (a) Stored memory entries and cumulative Medium/Hard solve progress, indexed by chronological solve order. "KB Entries" in the archived run denotes schema-memory entries. (b) Difficulty composition of solved problems in consecutive blocks of 50; the final bar contains the remaining problem.

## E MATCHED-PROTOCOL RESOURCE DIAGNOSTIC

To compare accuracy and inference consumption under the same evaluation stream, we report a matched-protocol diagnostic on the 511-problem LiveCodeBench manifest. The table reports aggregate request and token totals.

Table 9: Matched-protocol diagnostic on 511 LiveCodeBench problems. Total token usage is reported in millions. NoMem denotes Matched NoMemory and RBank denotes ReasoningBank.

| Metric | NoMem | RawReplay | ExpeL | A-MEM | RBank | VCST |
|---|---|---|---|---|---|---|
| Solved / 511 | 277 | 298 | 334 | 323 | 344 | 374 |
| Accuracy (%) | 54.21 | 58.32 | 65.36 | 63.21 | 67.32 | 73.19 |
| Model requests | 9,364 | 6,464 | 8,020 | 10,920 | 9,364 | 10,668 |
| Total tokens (M) | 12.663 | 14.994 | 14.946 | 14.131 | 14.931 | 14.082 |

On the 511-problem diagnostic set, VCST achieves the highest accuracy at 73.19%, improving over ReasoningBank by 5.87 percentage points. It also uses 14.082M tokens, below the other memory-based methods. These results show that verified schema transfer improves solution quality while using fewer tokens than the other memory-based methods.

## F PROTOCOL AND IMPLEMENTATION DETAILS

**Task streams and memory scope.** Each benchmark–backbone evaluation uses its own memory instance, initialized empty before processing the benchmark's fixed evaluation manifest. The manifest order is fixed before model calls and shared by all methods compared within that evaluation; memory is not shared across backbones or benchmarks. On LiveCodeBench, the scheduler cycles through unresolved tasks in this fixed manifest order. A schema obtained after verification can be used only on a later scheduled task visit.

**Generation and retrieval.** Unless a benchmark-specific evaluation protocol specifies otherwise, a VCST task visit uses five candidates at temperatures $\{0.6, 0.7, 0.8, 0.9, 1.0\}$. Failed visits rank candidates by available verifier feedback and repair the top two for up to three rounds. Retrieval first retains at most eight positive-overlap Jaccard candidates ($C_1$), then checks applicability and semantically selects at most $\kappa = 2$ schemas ($C_2$); an empty selection is permitted. Table 1 uses a single round of five-candidate generation per task, with no repair or deferred revisit. Memory updates made after the visit are used only for subsequent tasks. Figure 3 uses up to three repair rounds for the top two candidates and permits deferred visits under the round-robin protocol.

For Tables 2 and 3, each task is attempted once with memory fixed during solving. Correctness feedback is obtained after the answer is submitted and used solely for schema construction. It is not used to select candidates or revise the current answer; resulting memory updates apply only to subsequent tasks.

**Schema interfaces.** The LLM subroutines use structured interfaces rather than free-form memory writes. EXTRACT receives a task, an accepted solution, and its verification outcome, and returns `{insight, applicability, solution_template, tags, type, failure_modes, provenance}`. SYNTHESIZE receives verified entries and returns the same fields as a provisional schema; provisional schemas are not retrieved. REFINE receives schemas linked to verifier-passing solutions and returns consolidated schemas with explicit applicability conditions and verification records for all contributing sources. Query tagging returns a tag list, and reranking returns zero, one, or two applicable candidate indices. Candidate-generation prompts contain the task and selected schemas; repair prompts additionally contain verifier feedback.

**Baselines and model access.** Direct Inference uses a single greedy generation. On LiveCodeBench, $S^*$ uses 16 parallel samples and up to three debugging rounds; VCST uses the five-sample configuration above. The multi-task table reports one final scored outcome per task; CoT, Reflexion, CRITIC,

ExpeL, A-MEM, and ReasoningBank retain their respective prompting or memory-update mechanisms under the shared benchmark manifest and evaluator. Qwen-Coder denotes Qwen2.5-Coder-Instruct in the LiveCodeBench evaluation; the code-state evaluation in Table 3 uses Qwen3-32B. GPT backbones are accessed through their named API models, while Qwen-Coder, Qwen3-32B, and DeepSeek-R1-Distill backbones use their corresponding public releases. The new diagnostic tables use GPT-4o-mini through an OpenAI-compatible API; their run manifests record the requested model identifier, task manifest, and all counted requests and tokens.

**Memory-baseline configurations.** On LiveCodeBench, task-adapted ExpeL, A-MEM, and ReasoningBank share VCST's candidate-generation and repair settings: five initial candidates and at most three repair rounds for the top two candidates. Each generation or repair call has an output limit of 4,096 tokens. Up to two memory items are supplied to the solver. Each method starts with empty memory, uses the same fixed task order, and retains its own memory construction and update mechanism. The single-visit and deferred-revisit evaluations are distinguished in Section 4.1.

| Parameter | ExpeL | A-MEM | ReasoningBank |
|---|---|---|---|
| Retrieved items (maximum) | 2 | 2 | 2 |
| Memory extraction output limit | 400 tokens | 500 tokens | 500 tokens |
| Linked existing notes (maximum) | — | 3 | — |
| Link similarity threshold | — | 0.18 | — |
| Output limit per linked-note update | — | 300 tokens | — |
| MaTTS | — | — | Disabled |

For A-MEM, the link threshold is applied to bag-of-words cosine similarity. Reported model-call and token totals include generation, repair, memory operations, and model-call retries, following the accounting definition below.

**Diagnostic configurations.** The verification-gate diagnostic uses the same VCST retrieval and repair settings on its fixed 128-example stream. The matched-protocol resource diagnostic evaluates the six listed methods on the fixed 511-problem LiveCodeBench public-test manifest. These diagnostics complement the nine-backbone main result.

**Cost accounting.** Requests count all model calls incurred by each method, including auxiliary operations and any LLM-based verification. Tokens sum the input and output tokens consumed by these calls, including memory content supplied in prompts. API/P is the total number of model calls divided by the number of evaluated problems. Deterministic scheduling, tag matching, and test execution are outside these model-call and token metrics.

## G SOLVER PSEUDOCODE

Algorithm 1 gives the full VCST solver described in Section 3.3.
Here, $m = 5$, $R = 3$, $\kappa = 2$, and $\theta_l \in \{0.6, 0.7, 0.8, 0.9, 1.0\}$ are the full-protocol settings. $\mathrm{TopJaccard}_8$ retains up to eight positive-overlap entries. $\mathrm{Select}_\Phi$ jointly assesses applicability and relevance from the task and each entry's insight, conditions, and tags; it returns at most $\kappa$ schemas, or none. $Q$ holds the current candidates, and $v_i(Q)$ their verification feedback. $\mathrm{FirstPassing}$ selects the first passing candidate in generation order; $\mathrm{Top2}$ ranks by this feedback. The retrieved context $C_2$ remains fixed throughout each visit.

**Algorithm 1** VCST round-robin solver

**Require:** task set $T$, LLM $\Phi$, verification procedure $V$, API budget $B$

$K \leftarrow \varnothing$; $K_{\text{prov}} \leftarrow \varnothing$; $\sigma_i \leftarrow$ unsolved; $e_i \leftarrow 0$ for all $t_i \in T$
$H \leftarrow \varnothing$; $S \leftarrow (P, H)$, $P = \{(t_i, \sigma_i, e_i)\}_{i=1}^{|T|}$
**while** $B > 0$ and $\exists t_i : \sigma_i \neq$ solved **do**
  $a \leftarrow \Phi(\text{Summary}(S), K)$
  **if** $a =$ solve **then**
    $t_i \leftarrow \text{Scheduler}_{\text{RR}}(S)$
    $C_1 \leftarrow \text{TopJaccard}_8(K, t_i)$
    $C_2 \leftarrow \text{Select}_\Phi(C_1, t_i, \kappa)$
    $Q \leftarrow \{\Phi(\text{prompt}(t_i, C_2); \theta_l)\}_{l=1}^{m}$
    **if** $\exists s \in Q : \text{verify}(s, t_i, V) = 1$ **then**
      $s^* \leftarrow \text{FirstPassing}(Q, t_i, V)$
      $K \leftarrow K \cup \text{Extract}(\Phi, t_i, s^*, v_i(s^*))$; $\sigma_i \leftarrow$ solved
    **else**
      $Q \leftarrow \text{Top2}(Q, v_i(Q))$
      **for** $r = 1, \ldots, R$ **do**
        $Q \leftarrow \text{Repair}_\Phi(t_i, C_2, Q, v_i(Q))$
        **if** $\exists s \in Q : \text{verify}(s, t_i, V) = 1$ **then**
          $s^* \leftarrow \text{FirstPassing}(Q, t_i, V)$
          $K \leftarrow K \cup \text{Extract}(\Phi, t_i, s^*, v_i(s^*))$; $\sigma_i \leftarrow$ solved
          **break**
        **else**
          update $v_i(Q)$ for the repaired candidates
        **end if**
      **end for**
      **if** $\sigma_i \neq$ solved **then** $\sigma_i \leftarrow$ partial
      **end if**
    **end if**
    $e_i \leftarrow e_i + 1$
  **else if** $a =$ explore **then**
    $K_{\text{prov}} \leftarrow K_{\text{prov}} \cup \text{Synthesize}(\Phi, K, S)$
  **else if** $a =$ optimize **then**
    $K \leftarrow \text{Refine}(\Phi, K)$
  **end if**
  update $S$; $B \leftarrow B -$ (API calls used)
**end while**