CERTIJOIN: CERTIFIED EARLY TERMINATION FOR COMPOUND LLM PROGRAMS

Anonymous authors
Paper under double-blind review

ABSTRACT

LLM systems combine many model calls to vote on answers, search for solutions, and coordinate agents, yet their final output is often settled long before the last call returns. We introduce CertiJoin, a runtime verification framework that stops computation the moment it can no longer affect the program's declared output. Adapters declare the output rule and the possible results of each call, and a shared checking core determines when nested decisions and shared dependencies permit cancellation. Certificates bind this evidence to the current program and state. We prove exact output preservation for four join types and for local pruning in composed programs, with no probability calibration. In online nested code validation, the core cuts mean latency by 37.6% and tokens by 32.6% against full execution, preserves every paired success/failure outcome, and matches hand-written short-circuit logic to within 0.27%. In majority voting, CertiJoin preserves all 2,000 paired answers while finishing 84.5% of requests early, and in live serving it cuts mean latency by 27.6% with all 200 answers matching their reconstructed full-run references. CertiJoin also makes published stopping policies exact: when runner-up votes arrive first, they change up to 21.13% of answers, while their certified versions change none. Code and data: https://anonymous.4open.science/r/certijoin-3470.

1 INTRODUCTION

An LLM can answer a question by sampling several reasoning paths and choosing the most frequent final answer, a strategy known as *self-consistency* (Wang et al., 2023). Tree search scores intermediate reasoning steps (Yao et al., 2023a; Xie et al., 2023), while multi-agent debate lets model instances exchange and revise responses (Du et al., 2024b). These *compound LLM programs* coordinate model calls to produce a result (Chen et al., 2024; Zheng et al., 2024), and many of those calls stop mattering long before they finish. A vote may already have an unbeatable winner, and a failed component test can reject a code candidate before its remaining generation and tests finish (Chen et al., 2021).

Existing stopping policies exploit this through confidence or answer stability. Adaptive-Consistency (AC) uses posterior confidence (Aggarwal et al., 2023), Early-Stopping Self-Consistency (ESC) checks an agreement window (Li et al., 2024), and Dynasor's Certaindex uses answer entropy (Fu et al., 2025). These signals only estimate the final answer, so a policy that preserves every recorded answer can still flip answers once slow calls favor the runner-up. Such a flip is silent: the system runs faster but no longer returns the answer it was designed to compute. Guaranteeing the full-execution output requires showing that no unfinished call can change it, whichever calls finish first.

Consider 16 model calls voting on an answer. Four initial A answers give AC a confidence of $1 - 2^{-5} = 0.96875$, above its default threshold of 0.95. Yet the remaining 12 votes could all be B. After nine A answers, only seven votes remain and A must win. This margin is a deterministic stopping condition that holds whichever calls finish first (Fig. 1).

More generally, each model call or evaluation is a *branch*, and the rule that combines branch results is a *join*. A join may return a winning answer, an accepted candidate (Hao et al., 2023), or an ordered search beam. This declared output determines which computation can stop.

Real programs nest these joins, and the right stop then depends on details that hand-written logic fixes in advance. Rejecting candidate A cancels its private checks while a shared check continues for

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.

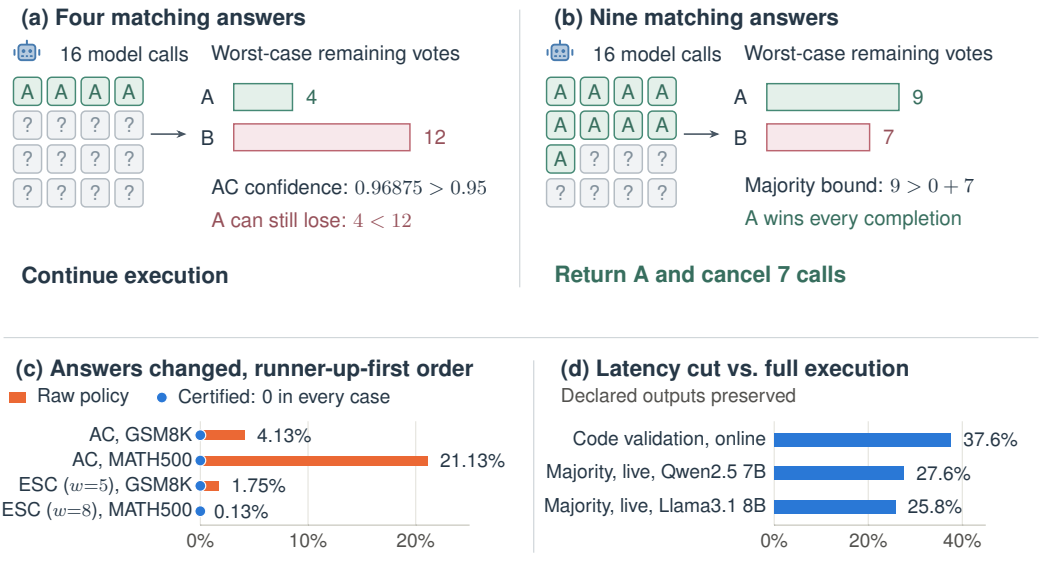

Figure 1: **Certified stopping preserves declared outputs and cuts latency.** (a, b) In a 16-call vote, four A answers exceed AC's confidence threshold, yet the remaining calls can elect B. Nine A answers fix the winner, so CertiJoin cancels seven calls. (c) Answers changed when runner-up votes arrive first; certification changes none. (d) Mean-latency reduction with declared outputs preserved.

B (Fig. 2). Returning the lowest-numbered candidate that passes its tests can stop after fewer checks than returning the highest-numbered one. Nine A votes settle a 16-branch vote but leave the winner unresolved with 24 branches. Hand-written stopping code must be rewritten whenever the sharing, the output rule, or the budget changes.

CertiJoin replaces this per-program engineering with runtime verification. An adapter declares the output expression and legal branch results, and a shared core checks when that output is fixed. Each proposed stop carries a certificate (JoinCert) that a verifier checks before any call is canceled. Policies remain free to use confidence or prediction, while the guarantee rests only on completed results and declared rules. Table 1 contrasts this design with existing stopping approaches.

Table 1: Early-stopping approaches.

Approach	No per-rule stop logic	Exact output	Exact pruning
AC, ESC	✗	✗	✗
Certaindex	✓	✗	✗
Hand-written	✗	✓	✓
CertiJoin	✓	✓	✓

We make three contributions:

- We cast early termination of compound LLM programs as runtime verification and prove exact output preservation for four join types and for local pruning in composed programs.
- We design one checking core and a certificate interface, JoinCert, that separate what a program outputs from when it may stop, so a new program needs only an output declaration.
- We show that one unchanged core serves search, debate, and code validation, matching hand-written short circuit online and staying exact when output rules or budgets change. In majority voting, it preserves all 2,000 paired answers, cuts live latency by 27.6%, and removes the answer changes that adversarial orders cause in published policies.

2 Program Model and Join Semantics

To state what an early stop must preserve, we model execution at a join as $P = (x, \mathcal{C}, \mathcal{J}, \mathcal{B}_0)$ with input x, controller $\mathcal{C}$, combination rule $\mathcal{J}$, and initial branches $\mathcal{B}_0$. The controller forks and joins calls from this initial set, recording outputs and execution metadata in state S_t (Welleck et al., 2024). A shortcut a changes this state by canceling work or returning early.

A shortcut must preserve the program's *declared output*, which each join type defines. MajorityJoin returns the winning answer. FirstSuccessJoin and VerifierAcceptJoin decide whether an accepted candidate, called a *witness*, exists. AllRequiredJoin requires every subtask to complete. Applications that use a particular candidate, ordered list, or later-round message include it in their declared output. For composed joins, an expression E represents this output. Its leaves are branch results, with one variable per shared result and a domain D_b covering every legal value of leaf b.

Let $\mathrm{Exec}_{\mathcal{J}}(P, S; \omega)$ denote the declared output and committed effects after complete execution. The state fixes the covered branches, answer normalization (the *canonicalizer*), tie rule, and test/verifier versions. A replay specification ω fixes one admissible completion and its nondeterminism. The original and shortcut runs share ω, and preservation requires agreement for every such completion:

Definition 1 (Semantic preservation). A shortcut preserves join $\mathcal{J}$ when

$$\mathrm{SemPres}_{\mathcal{J}}(P, S_t, a) \iff \forall \omega \in \Omega(P, S_t),\ \mathrm{Exec}_{\mathcal{J}}(P, S_t; \omega) = \mathrm{Exec}_{\mathcal{J}}\big(P, a(S_t); \omega\big), \quad (1)$$

where $\Omega(P, S_t)$ is the set of replay specifications consistent with the declared rules and completed results in S_t.

3 CertiJoin

CertiJoin enforces Definition 1 at runtime. The adapter declares the output expression, the policy proposes actions, a certificate generator packages the evidence for each proposal, and the verifier checks it before the executor cancels calls or returns. We start with the shared core that decides when a composed result is fixed, then show how the runtime acts on that decision safely.

3.1 Checking Composed Results

Code search combines decisions at two levels. Candidate c qualifies when it passes its component tests $g_{c,j}$, integration test h_c, and any shared checks U_c. With $g_c = \bigwedge_j g_{c,j}$, its acceptance and the search outcome are

$$q_c = g_c \wedge h_c \wedge \bigwedge_{b \in U_c} y_b, \qquad F = \bigvee_c q_c.$$

A single failed component fixes q_c to false, so all work used only by that candidate can stop at once. Figure 2 uses $1/0$ for true/false and shows three decisions. (1) A component of A fails, so its private integration check h_A can stop. (2) B still needs shared check u, so u remains live and execution continues with $F = q_B \vee q_C$. (3) B passes all its checks, fixing $F = 1$ so success returns and C is canceled. The same core verifies each decision from the declared expression and completed results.

Expressions and possible results. The adapter translates the program's output rule into sums, votes, ordered top-k selection, and conditionals. It assigns leaf identities and domains, routing completed values to the core for stopping checks. The acceptance test or parser fixes each domain before generation: a test yields true or false, and a rating parser yields a declared score. Every use of a shared result refers to the same leaf, which is replaced by its value on completion.

Checking whether the result is fixed. The core substitutes completed values into the expression and evaluates its possible completions. Finite conditionals, candidate selection, and votes use the same enumeration procedure. The check exits as soon as two assignments disagree; if all agree, their common value certifies the result. This procedure also applies to subexpressions, allowing a candidate's outcome to be fixed while the rest of the program continues.

Search scores, such as Tree of Thoughts (ToT) ratings summed across calls, use an affine bound. For $s_i - s_j = \beta + \sum_b \alpha_b y_b$ and completed leaves O_t, the minimum difference is

$$L_{ij} = \beta + \sum_{b \in O_t} \alpha_b y_b + \sum_{b \notin O_t} \min_{v \in D_b} \alpha_b v. \quad (2)$$

Repeated uses of a leaf contribute to the same coefficient. Thus $s_A = u + 2$ and $s_B = u + 1$ have a fixed difference of one even while u is unfinished. A program returning the top-scoring identity can return A immediately, while returning the scores still requires u. The core certifies an ordered top-k result once the bounds fix every comparison that order needs under the declared tie rule.

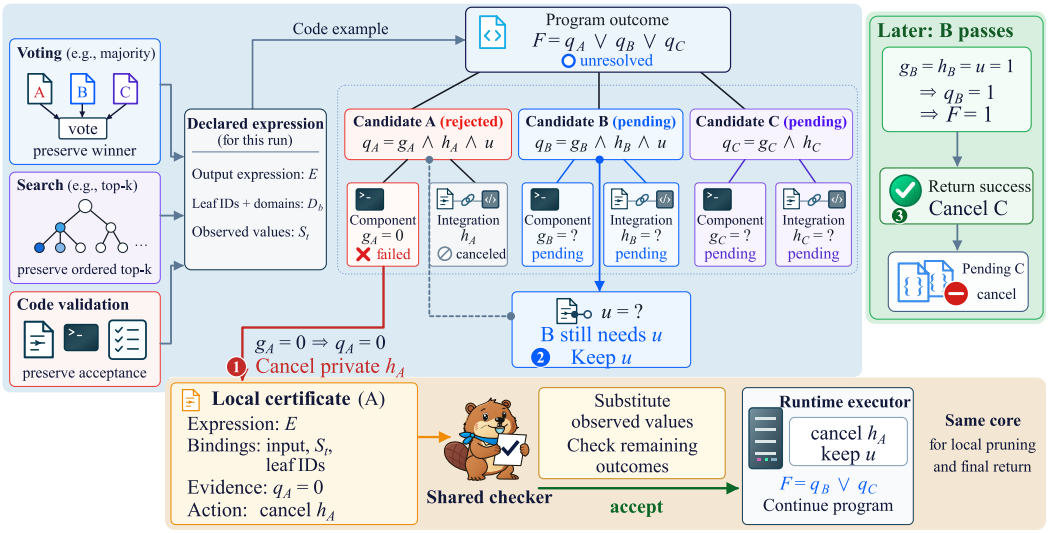

Figure 2: One checking core for local pruning and program termination. The core (1) cancels A's private check after rejection, (2) keeps the shared check for B, and (3) returns success and cancels C once B passes. Certificates bind each action to the declared expression and current state.

From a fixed result to cancellation. A certificate binds the determined value to the expression, domains, input context, and current state. The executor substitutes that value and cancels calls with no remaining consumers. Shared calls stay live while another expression needs them. Once the root is fixed, the executor returns its value and cancels the remaining work covered by the certificate.

Because pruning follows the declared output, a new output rule changes the stop without new code. Returning the lowest-numbered qualifying candidate requires only that lower-numbered ones have failed: on the recorded DataStatistics code task, this takes three component checks, while returning the highest-numbered one takes twelve on the same trace. The same core handles both.

3.2 Verification and Execution

Acting on a fixed result also requires that the evidence still hold when the action commits. After a branch completes, CertiJoin updates S_t and queries policy Π using completed outputs and runtime metadata. The generator constructs certificate c for proposed action a. The verifier checks its type and state bindings, then recomputes the acceptance condition. If $\mathrm{VerifyCert}(P, S_t, a, c) = 1$, the executor applies the action atomically to that state version. Local pruning updates the remaining expression and execution continues. A terminal action returns the verified output and cancels its covered work, as Algorithm 1 shows.

The certificate's *state bindings* identify the covered branches, input, acceptance-test version, and execution environment (App. B.1). Recomputing these bindings prevents evidence from a different branch budget or test environment from authorizing an action, as in the 16-versus-24-branch example.

A rejected proposal costs only the check, and execution continues. The audit log records every proposal with its state, so replay pinpoints which proposals would have changed the output.

3.3 Deterministic Certificates

Four common join types need no enumeration: closed-form conditions certify their declared outputs directly (Table 2). Majority experiments use MajorityCert, while search, debate, and nested code use the general expression core; both share the same binding and action checks.

Algorithm 1: Certified local pruning and terminal return.

Input: Program P, declared branch universe, action policy Π
Output: Declared output z and audit log H

Initialize runtime state S and empty log H
while *unresolved branches remain* **do**
  $(b, y) \leftarrow \textsc{NextCompletion}(P)$
  $S \leftarrow \textsc{UpdateState}(S, b, y)$
  $a \leftarrow \Pi(P, S)$ ▷ observed prefix only
  if $a = \perp$ **then**
    continue
  end if
  $c \leftarrow \textsc{BuildCert}(P, S, a)$
  $v \leftarrow \textsc{VerifyCert}(P, S, a, c)$ ▷ type, bindings, join condition
  Record state hash, a, c, v, and rejection reason in H
  if $v = 1$ **then**
    $S \leftarrow \textsc{ApplyCertifiedAction}(P, S, a, c)$
    if a *is terminal* **then**
      return $(\textsc{VerifiedOutput}(S), H)$
    end if
    continue ▷ local pruning keeps the program running
  end if
end while
return $(\textsc{CompleteJoin}(P, S), H)$

Table 2: Deterministic conditions and declared outputs under Assumption 1. AllRequired uses the proposed post-action sets.

Certificate	Acceptance condition	Declared output
MajorityCert	$m_1 > m_2 + r_{\max}$	Winning answer class.
FirstSuccessCert	$\chi(x, y_b) = 1$	Existence of an accepted candidate.
AllRequiredCert	$\mathcal{R}_t \subseteq \mathcal{C}'_t \cup \mathcal{D}'_t$ $\mathcal{R}_t \cap \mathcal{X}'_t = \varnothing$	Completion of all required subtasks.
VerifierAcceptCert	$v_\nu(x, y_b, \pi_b) = 1$	Existence of an accepted witness.

MajorityJoin. Let m_1 and m_2 be the leader and runner-up vote counts after canonicalization, with at most $r_{\max}$ votes unresolved. MajorityCert accepts when

$$m_1 > m_2 + r_{\max}. \tag{3}$$

Even if every unresolved vote favors the runner-up, the leader remains ahead. Appendix A.2 proves this condition and extends it to class-specific residual bounds.

FirstSuccessJoin and VerifierAcceptJoin. For FirstSuccessJoin, a registered acceptance test χ checks a completed candidate, as in execution-validated code search (Chen et al., 2021). VerifierAcceptJoin uses a witness predicate $v_\nu(x, y_b, \pi_b)$ for supplied evidence π_b. Each certificate binds the candidate, input, and predicate version to the current state. Acceptance establishes that a witness exists; rules that select a particular candidate go through the expression core of Sec. 3.1.

AllRequiredJoin. Let $\mathcal{R}_t$ be the required branches and $\mathcal{C}'_t$, $\mathcal{D}'_t$, and $\mathcal{X}'_t$ the completed, deferred, and deleted sets after an action. The condition in Table 2 retains required work through completion or recoverable deferral. Deferred work must resume, and terminal return requires $\mathcal{R}_t \subseteq \mathcal{C}'_t$.

3.4 Preservation Guarantee

Assumption 1 (Verifier and execution assumptions). The guarantees require these conditions:

(i) The certificate binds the join operator, tie rule, canonicalizer, acceptance predicates, and certificate verifier.
(ii) Verification and action commit use the same state version, so intervening completions cannot invalidate the evidence. The executor applies only verified substitutions, return values, and cancellation targets.
(iii) Replay specifications fix the sources of nondeterminism relevant to the declared join semantics.
(iv) Effects of cancelable branches remain isolated and are discarded on cancellation. Existing commits remain unchanged, and later commits depend only on the preserved declared output and fixed input context.
(v) For AllRequiredJoin, deferred work remains recoverable and completes under every admissible continuation. Terminal actions have no unfinished required work.

Theorem 1 (Soundness of deterministic certificates). For each join in Table 2 with the declared output specified in Sec. 2, Assumption 1 and $\mathrm{VerifyCert}(P, S_t, a, c) = 1$ imply that, for every $\omega \in \Omega(P, S_t)$,

$$\mathrm{Exec}_{\mathcal{J}}(P, S_t; \omega) = \mathrm{Exec}_{\mathcal{J}}\big(P, a(S_t); \omega\big). \tag{4}$$

App. A.1 proves this result and the following composition property, including committed effects.

Proposition 1 (Composed result preservation). Let E be a deterministic declared output expression whose leaf domains cover all legal completions, with one variable per shared branch result. If a subexpression has the same value v under every assignment consistent with S_t, replacing it by v preserves E. Under Assumption 1, canceling calls on which no remaining declared output depends preserves program semantics.

3.5 Backend Cancellation

CertiJoin dispatches certified cancellations to the serving backend: the majority controller closes vLLM streams after a verified stop, and the code executor sends SGLang abort requests after candidate rejection or a final decision. Each cancellation is matched to the backend's admission and abort records, so reported savings reflect generation that actually stopped. Because branches can share prefix blocks, a canceled branch's key–value (KV) blocks are freed once no other branch or cache holds them (Fig. A3).

4 Experiments

4.1 Experimental Setup

Our experiments answer four questions: whether one checking core can replace hand-written stopping logic across programs (Sec. 4.2), whether certified stopping keeps every answer while saving computation offline and in live serving (Sec. 4.3), whether certification makes existing stopping policies exact (Sec. 4.4), and whether verification is cheap and reliable (Sec. 4.5).

We compare CertiJoin with full execution, published stopping policies, and Dedicated short circuit, which hand-codes the same output rule and safe stopping conditions. Majority replay runs $N = 16$ branches with Qwen2.5 7B/14B/32B and Llama3.1 8B on 200 questions per model from GSM8K and MATH500 and 100 from MMLU (Cobbe et al., 2021; Hendrycks et al., 2021b;a). Certaindex runs on a Llama3.1 8B trace with $N = 30$ and 200 GSM8K questions. HumanEval and MBPP test first-success search (Chen et al., 2021; Austin et al., 2021), and an AllRequired workload tests that unsafe deletions are rejected.

Fixed-trace comparisons replay identical recorded branch outputs and count disagreements with full execution; speedup is the ratio of mean full-execution cost to mean policy cost, including certificates. In majority replay, a policy proposes a stop once the leading answer reaches half of the valid observed answers ($\tau = 0.5$), and CertiJoin certifies these proposals.

4.2 Replacing Hand-Written Stopping Logic

One unchanged core replaces hand-written stopping logic across three program families. It preserves every declared output of ToT search, final-round debate aggregation, and nested code validation, and

agrees with independent dedicated implementations at all 6,567 partial states (Table 3). Each adapter specifies only expressions and domains; none contains stopping code.

Keeping stopping logic out of adapters also removes a maintenance risk. A hand-written stop hard-codes its output rule and budget, so a change that never reaches it leaves a stale stop that silently alters outputs. A stop written to pick the lowest-numbered qualifying candidate, left in place after the rule flips to the highest or the reverse, returns the wrong candidate in 3 of the 5 code classes with a qualifying candidate, and a 16-vote margin check changes 2.5% of GSM8K answers once the budget grows to 30. With CertiJoin, such changes only update the adapter's declaration, and the verifier rejects every stale stop that would alter the output (App. D.1).

Reuse does not cost efficiency. In serial replay on 14 code-validation classes, CertiJoin skips the unfinished components of rejected candidates and stops once the existence of a qualifying candidate is settled. With fixed candidate order, component checks fall from 148 to 71 and recorded generation from 56,036 to 26,197 tokens. Local pruning raises token savings from 29.91% with program-level termination to 53.25%. Hand-written Dedicated short circuit makes exactly the same decisions (App. D.2).

Table 3: One unchanged checking core on recorded traces. Mismatch counts partial states where it disagrees with a dedicated checker.

Program	Configs	Preserved ↑	Mismatch ↓
ToT, 100 questions	400	**400/400**	**0**
Debate, 24 questions	96	**96/96**	**0**
Code, 18 classes	108	**108/108**	**0**

The savings carry over to online execution. We rerun these 14 ClassEval-derived classes (Du et al., 2024a) with fresh generation and the original integration tests. All methods admit calls in candidate order with concurrency four and two repetitions per class, and tokens are counted through aborted streams (Fig. 3).

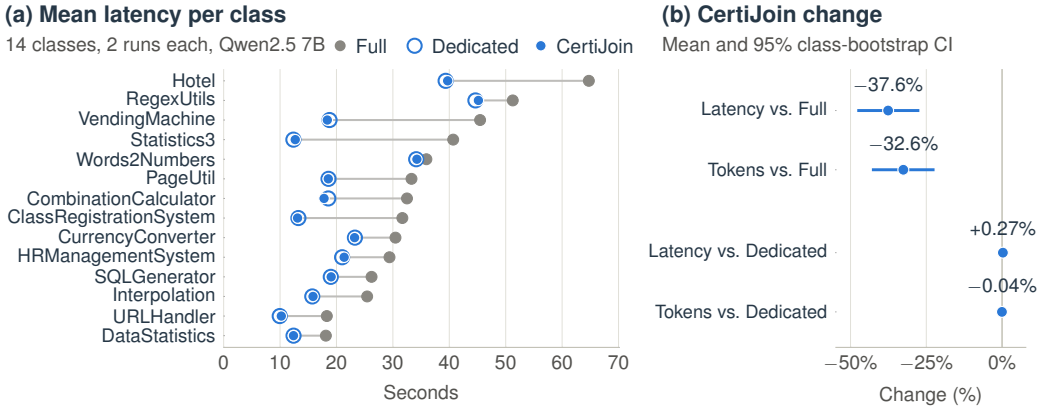

Figure 3: Online nested code validation (Qwen2.5 7B). (a) CertiJoin (dot) matches hand-written Dedicated short circuit (ring) and is faster than Full in every class. (b) Paired latency and token changes with 95% class-bootstrap intervals.

CertiJoin runs faster than Full in every class, cutting mean latency by 37.6% and tokens by 32.6% while preserving all 28 paired success/failure outcomes. The unchanged core matches hand-written Dedicated short circuit to within 0.27% (95% interval: -0.62 to $+0.94$%). Most savings come from the local decisions of Fig. 2: rejected candidates are pruned while the program keeps running.

4.3 Preserving Answers While Saving Computation

Majority voting shows the same pattern across four models and three benchmarks. On the fixed 2,000-request slice, CertiJoin preserves every full-run answer and the 74.65% benchmark accuracy. Accepting the same proposals without certification changes 16.60% of answers and reduces accuracy by 4.25 points. Even a conservative rule that waits for half the branches changes 1.15% of answers (App. D.4). At comparable replay cost, a fixed 13-branch budget on MATH500 gives 1.23× speedup

and changes 7.88% of answers, while CertiJoin gives 1.25× with all answers preserved (App. D.6). The savings are broad: CertiJoin finishes 84.5% of requests earlier than full execution, with a median saving of 42.2% (Table 4). They also adapt to difficulty: certified runs skip about 40% of the 16 branches on GSM8K and MMLU and keep more on MATH500, where answers stay split longer (App. D.3).

Table 4: CertiJoin's request-level savings in paired replay; speedups span the four models.

Dataset	Requests	Faster requests ↑	Median saving ↑	Speedup ↑
GSM8K	800	97.3%	43.4%	**1.53–1.73×**
MATH500	800	64.9%	26.6%	**1.08–1.28×**
MMLU	400	98.3%	43.3%	**1.55–1.75×**
All	**2,000**	**84.5%**	**42.2%**	—

Live serving. The gains also hold on a live backend. The live controller streams all 16 branches and closes unfinished streams after certification. Qwen2.5 7B serves 100 GSM8K questions at 30 queries per minute (QPM) in a counterbalanced crossover, giving 200 executions per policy with shared arrivals and seeds. Latency includes queueing (App. C.4).

Table 5: Live serving with backend cancellation, Qwen2.5 7B at 30 QPM, 200 request executions per policy.

	Latency (s) ↓			
Policy	**Mean**	**p50**	**p95**	**p99**
Full	29.70	31.06	40.79	42.52
CertiJoin	**21.50**	**22.22**	**30.66**	**32.00**
Reduction	**27.6%**	**28.5%**	**24.8%**	**24.7%**

All 200 certified answers match their reconstructed full joins, and both policies reach 94% accuracy. The controller returns early for 99.5% of requests, after 9.44 branches on average. Because the join no longer waits for its slowest calls, tail latency falls nearly as much as the mean, by 24.8% at p95. Backend records confirm all 1,304 cancellations of admitted requests, and probes show zero running requests and zero KV allocation by the first 0.25-second poll after cancellation.

The gains hold across loads, concurrency limits, and a second model, with every reconstructed join preserved and p95 latency down by up to 25.5%; Llama3.1 8B cuts mean latency by 25.8%. Paired serving replay at 30–120 QPM gives 1.37–1.54× speedups with unchanged answers (App. D.8).

4.4 Certifying Existing Stopping Policies

Certification also repairs existing heuristics: wrapping AC, ESC, and Certaindex makes each of them exact. We run AC and ESC on the $N = 16$ majority traces and Certaindex on the $N = 30$ trace (App. C.3).

In recorded order, AC and ESC happen to preserve every full join, and certified AC runs 1.658× faster than full execution on GSM8K (Table A10). The picture changes once completion order does. Figure 1(c) keeps branch outputs fixed but lets runner-up votes arrive first: AC then changes 21.13% of MATH500 answers, while its certified version preserves them all.

The sensitivity remains at equal cost: AC, ESC, answer-stability, and vote-margin rules matched to CertiJoin's recorded-order cost still change 1.75–5.13% of MATH500 answers under the runner-up-first order (App. D.6). Even random completion orders flip 88 of AC's 80,000 MATH500 evaluations; certification flips none. At equal cost, only certified stopping keeps every answer.

On the $N = 30$ GSM8K trace, Certaindex changes 9.50% of answers under the runner-up-first order and none when certified, while saving 25.38% of tokens with repeated five-branch checks.

4.5 Verification Cost and Reliability

These guarantees cost milliseconds: verification takes under 0.1% of mean latency in both majority voting and nested code. Invoked after every completion on the 2,000-request slice, CertiJoin spends 1.355 ms per request and stops at exactly the same positions as a direct margin check. In online nested code, the shared core adds 18.50 ms per program, 0.086% of the 21.55-second mean latency.

Witness certificates rerun the acceptance test yet keep HumanEval and MBPP search near native speed, preserving every success outcome; returning the first candidate untested changes 4–17% of them.

The verifier is also reliable: it matches reference decisions on all 100,000 random states and independent enumeration on 4,378 partial states from 17 program variants, and it rejects all 4,716 answer-changing majority proposals across the registered budgets and thresholds. Binding tests reject changed, unregistered, and tampered state, and an 800-program HumanEval/MBPP audit reproduces the reported outcomes (App. B.1).

5 Related Work

Adaptive inference and stopping. Test-time compute methods allocate effort through search, revision, and budget control (Snell et al., 2025; Muennighoff et al., 2025), as well as early exit, branch allocation, predictive budgets (Yang et al., 2025; Wang et al., 2025; Jiang et al., 2025; Brown et al., 2026), and self-verification signals (Zhang et al., 2025). Recent work certifies stopping against a preservation target: MARS uses intermediate probes for high-probability full-budget vote preservation, while CITE certifies a distributional mode under adaptive stopping (Chen et al., 2026; Ota et al., 2026). CertiJoin instead makes preservation of the declared program output exact, so any confidence or stability signal can drive proposals while certificates authorize execution changes.

Programmable LLM runtimes. Parrot exposes application dataflow to LLM services, and LLM-Compiler plans parallel function calls (Lin et al., 2024; Kim et al., 2024). SGLang, PIE, Autellix, and Atomix support structured or agent execution (Zheng et al., 2024; Gim et al., 2025; Luo et al., 2025; Mohammadi et al., 2026). CertiJoin adds verified cancellation at this application level and dispatches it through vLLM or SGLang, whose backends manage batching and KV reuse (Yu et al., 2022; Kwon et al., 2023; Ye et al., 2024). Speculative Actions validates predicted sequential transitions (Ye et al., 2026), whereas our certificates verify joins over concurrent branches.

Evaluation and aggregation. Short-circuit and lazy evaluation avoid work not needed for an output. Partial evaluation specializes a program using known inputs (Jones et al., 1993), and threshold algorithms determine top-k results from bounds on unseen scores (Fagin et al., 2003). CertiJoin applies these principles to declared LLM output expressions, binding the evidence to an execution state and retaining shared calls when other outputs still need them, so one check stays exact as output rules, budgets, and sharing change.

Proof-carrying and certified reasoning. JoinCert follows proof-carrying code by attaching checkable evidence to runtime actions (Necula, 1997). Certified Self-Consistency and related methods certify answer reliability or generated artifacts (Cordero-Encinar & Duncan, 2025; Li et al., 2025), whereas CertiJoin certifies that stopping leaves the declared program output unchanged.

6 Conclusion

CertiJoin prunes computation that can no longer affect the declared output and returns once that output is fixed. One shared core checks nested decisions and shared dependencies across search, debate, and code validation, preserving every declared output while matching hand-written short circuit online. In majority voting, it cuts mean and tail latency in live serving and keeps every answer unchanged even under adversarial completion orders. Because proposals are separated from verification, any heuristic or learned stopping policy can now be made exact, turning early termination from per-program engineering into a checked runtime service.

AI USE STATEMENT

In this work, we used LLMs to assist with writing and experiments. The authors wrote the paper, and LLMs helped polish and revise the text. The authors designed the experiments and analyzed the results, while LLMs assisted with implementation and execution under the authors' direction and with their direct participation. We did not use LLMs to generate research ideas. We have reviewed all AI-assisted text and experimental work and take full responsibility for the final content of this work, including all text, claims, results, and artifacts.

ETHICS STATEMENT

Our experiments use public benchmarks and recorded model outputs, with no human-subject study or collection of private user data. Generated code is tested in temporary subprocesses or a restricted sandbox. We report output preservation separately from benchmark accuracy, alongside latency and token costs.

REPRODUCIBILITY STATEMENT

Section 3 and Algorithm 1 describe verification and execution, with proofs in App. A. Section 4 and Apps. C–D specify the datasets, model configurations, branch budgets, stopping policies, and measurement protocols. Certificate fields and paired uncertainty estimates are documented in the appendices. The abstract links to the anonymous code and data repository.

REFERENCES

Pranjal Aggarwal, Aman Madaan, Yiming Yang, and Mausam. Let's sample step by step: Adaptive-consistency for efficient reasoning and coding with LLMs. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 12375–12396, 2023.

Amey Agrawal, Ashish Panwar, Jayashree Mohan, Nipun Kwatra, Bhargav S Gulavani, and Ramachandran Ramjee. SARATHI: Efficient LLM inference by piggybacking decodes with chunked prefills. *arXiv preprint arXiv:2308.16369*, 2023.

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

Katrina Brown, Aneesh Muppidi, and Rana Shahout. Predictive scheduling for efficient inference-time reasoning in large language models. *arXiv preprint arXiv:2602.01237*, 2026.

Lingjiao Chen, Jared Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. Are more LLM calls all you need? Towards the scaling properties of compound AI systems. In *Advances in Neural Information Processing Systems*, volume 37, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/51173cf34c5faac9796a47dc2fdd3a71-Abstract-Conference.html.

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

Wenbo Chen, Puheng Li, Mengyang Liu, Weijie Su, and Tianpei Xie. MARS: Margin-adversarial risk-controlled stopping for parallel LLM test-time scaling. *arXiv preprint arXiv:2606.12935*, 2026. URL https://arxiv.org/abs/2606.12935.

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

Paula Cordero-Encinar and Andrew B Duncan. Certified self-consistency: Statistical guarantees and test-time training for reliable reasoning in LLMs. *arXiv preprint arXiv:2510.17472*, 2025.

Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. Evaluating large language models in class-level code generation. In *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering*, 2024a. doi: 10.1145/3597503.3639219. URL https://mingwei-liu.github.io/assets/pdf/ICSE2024ClassEval-V2.pdf.

Yilun Du, Shuang Li, Antonio Torralba, Joshua B. Tenenbaum, and Igor Mordatch. Improving factuality and reasoning in language models through multiagent debate. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 11733–11763. PMLR, 2024b. URL https://proceedings.mlr.press/v235/du24e.html.

Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003. doi: 10.1016/S0022-0000(03)00026-6. URL https://arxiv.org/abs/cs/0204046.

Yichao Fu, Junda Chen, Siqi Zhu, Zheyu Fu, Zhongdongming Dai, Yonghao Zhuang, Yian Ma, Aurick Qiao, Tajana Rosing, Ion Stoica, and Hao Zhang. Efficiently scaling LLM reasoning programs with Certaindex. In *Advances in Neural Information Processing Systems*, volume 38, 2025. URL https://papers.nips.cc/paper_files/paper/2025/hash/d037fd021c9aace128b8ce25001cdb6c-Abstract-Conference.html.

In Gim, Zhiyao Ma, Seung-seob Lee, and Lin Zhong. PIE: A programmable serving system for emerging LLM applications. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pp. 415–430, 2025.

Shibo Hao, Yi Gu, Haodi Ma, Joshua Hong, Zhen Wang, Daisy Wang, and Zhiting Hu. Reasoning with language model is planning with world model. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 8154–8173, 2023.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. In *International Conference on Learning Representations*, 2021a.

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 2021 Datasets and Benchmarks Track*, 2021b.

Weifan Jiang, Rana Shahout, Yilun Du, Michael Mitzenmacher, and Minlan Yu. Intra-request branch orchestration for efficient LLM reasoning. *arXiv preprint arXiv:2509.24957*, 2025.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *International Conference on Learning Representations*, 2024.

Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993. URL https://studwww.itu.dk/people/sestoft/pebook/.

Sehoon Kim, Suhong Moon, Ryan Tabrizi, Nicholas Lee, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. An LLM compiler for parallel function calling. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pp. 24370–24391. PMLR, 2024. URL https://proceedings.mlr.press/v235/kim24y.html.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Tangrui Li, Pei Wang, Hongzheng Wang, Christian Hahm, Matteo Spatola, and Justin Shi. PCRLLM: Proof-carrying reasoning with large language models under stepwise logical constraints. *arXiv preprint arXiv:2511.08392*, 2025.

Yiwei Li, Peiwen Yuan, Shaoxiong Feng, Boyuan Pan, Xinglin Wang, Bin Sun, Heda Wang, and Kan Li. Escape sky-high cost: Early-stopping self-consistency for multi-step reasoning. In *International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=ndR8Ytrzhh.

Chaofan Lin, Zhenhua Han, Chengruidong Zhang, Yuqing Yang, Fan Yang, Chen Chen, and Lili Qiu. Parrot: Efficient serving of LLM-based applications with semantic variable. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 929–945. USENIX Association, 2024. URL https://www.usenix.org/conference/osdi24/presentation/lin-chaofan.

Xiao Liu, Hao Yu, Hanchen Zhang, Yifan Xu, Xuanyu Lei, Hanyu Lai, Yu Gu, Hangliang Ding, Kaiwen Men, Kejuan Yang, et al. AgentBench: Evaluating LLMs as agents. In *International Conference on Learning Representations*, 2024.

Yuhan Liu, Yihua Cheng, Jiayi Yao, Yuwei An, Xiaokun Chen, Shaoting Feng, Yuyang Huang, Samuel Shen, Rui Zhang, Kuntai Du, et al. LMCache: An efficient KV cache layer for enterprise-scale LLM inference. *arXiv preprint arXiv:2510.09665*, 2025.

Michael Luo, Xiaoxiang Shi, Colin Cai, Tianjun Zhang, Justin Wong, Yichuan Wang, Chi Wang, Yanping Huang, Zhifeng Chen, Joseph E Gonzalez, et al. Autellix: An efficient serving engine for LLM agents as general programs. *arXiv preprint arXiv:2502.13965*, 2025.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, et al. SpecInfer: Accelerating generative large language model serving with tree-based speculative inference and verification. *arXiv preprint arXiv:2305.09781*, 2023.

Bardia Mohammadi, Nearchos Potamitis, Lars Klein, Akhil Arora, and Laurent Bindschaedler. Atomix: Timely, transactional tool use for reliable agentic workflows. *arXiv preprint arXiv:2602.14849*, 2026.

Niklas Muennighoff, Zitong Yang, Weijia Shi, Xiang Lisa Li, Li Fei-Fei, Hannaneh Hajishirzi, Luke Zettlemoyer, Percy Liang, Emmanuel Candès, and Tatsunori B Hashimoto. s1: Simple test-time scaling. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pp. 20275–20321, 2025.

George C Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 106–119, 1997.

Hirofumi Ota, Naoto Iwase, Yuki Ichihara, Junpei Komiyama, and Masaaki Imaizumi. CITE: Anytime-valid statistical inference in LLM self-consistency. *arXiv preprint arXiv:2605.05873*, 2026. URL https://arxiv.org/abs/2605.05873.

Pratyush Patel, Esha Choukse, Chaojie Zhang, Aashaka Shah, Íñigo Goiri, Saeed Maleki, and Ricardo Bianchini. Splitwise: Efficient generative LLM inference using phase splitting. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*, pp. 118–132. IEEE, 2024.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. FlexGen: High-throughput generative inference of large language models with a single GPU. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 31094–31116. PMLR, 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. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/1b623663fd9b874366f3ce019fdfdd44-Paper-Conference.pdf.

Vikranth Srivatsa, Zijian He, Reyna Abhyankar, Dongming Li, and Yiying Zhang. Preble: Efficient distributed prompt scheduling for LLM serving. *arXiv preprint arXiv:2407.00023*, 2024.

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

Yuhang Wang, Youhe Jiang, Bin Cui, and Fangcheng Fu. Thinking short and right over thinking long: Serving LLM reasoning efficiently and accurately. *arXiv preprint arXiv:2505.13326*, 2025.

Sean Welleck, Amanda Bertsch, Matthew Finlayson, Hailey Schoelkopf, Alex Xie, Graham Neubig, Ilia Kulikov, and Zaid Harchaoui. From decoding to meta-generation: Inference-time algorithms for large language models. *arXiv preprint arXiv:2406.16838*, 2024.

Yuxi Xie, Kenji Kawaguchi, Yiran Zhao, James Xu Zhao, Min-Yen Kan, Junxian He, and Michael Qizhe Xie. Self-evaluation guided beam search for reasoning. *Advances in Neural Information Processing Systems*, 36:41618–41650, 2023.

Chenxu Yang, Qingyi Si, Yongjie Duan, Zheliang Zhu, Chenyu Zhu, Qiaowei Li, Minghui Chen, Zheng Lin, and Weiping Wang. Dynamic early exit in reasoning models. *arXiv preprint arXiv:2504.15895*, 2025.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Tom Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of Thoughts: Deliberate problem solving with large language models. *Advances in Neural Information Processing Systems*, 36:11809–11822, 2023a.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R. Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023b.

Lu Ye, Ze Tao, Yong Huang, and Yang Li. ChunkAttention: Efficient self-attention with prefix-aware KV cache and two-phase partition. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 11608–11620, 2024.

Naimeng Ye, Arnav Ahuja, Georgios Liargkovas, Yunan Lu, Kostis Kaffes, and Tianyi Peng. Speculative actions: A lossless framework for faster agentic systems. In *International Conference on Learning Representations*, 2026. URL https://proceedings.iclr.cc/paper_files/paper/2026/file/fb279d7a134ad6fead7953ac7fd77fc4-Paper-Conference.pdf.

Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 521–538, 2022.

Anqi Zhang, Yulin Chen, Jane Pan, Chen Zhao, Aurojit Panda, Jinyang Li, and He He. Reasoning models know when they're right: Probing hidden states for self-verification. *arXiv preprint arXiv:2504.05419*, 2025.

Lianmin Zheng, Liangsheng Yin, Zhiqiang Xie, Chuyue Sun, Jeff Huang, Cody H Yu, Shiyi Cao, Christos Kozyrakis, Ion Stoica, Joseph E Gonzalez, et al. SGLang: Efficient execution of structured language model programs. *Advances in Neural Information Processing Systems*, 37:62557–62583, 2024.

Yinmin Zhong, Shengyu Liu, Junda Chen, Jianbo Hu, Yibo Zhu, Xuanzhe Liu, Xin Jin, and Hao Zhang. DistServe: Disaggregating prefill and decoding for goodput-optimized large language model serving. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pp. 193–210, 2024.

Andy Zhou, Kai Yan, Michal Shlapentokh-Rothman, Haohan Wang, and Yu-Xiong Wang. Language agent tree search unifies reasoning, acting, and planning in language models. *arXiv preprint arXiv:2310.04406*, 2023.

A Formal Details and Proofs

A.1 Proofs of Theorem 1 and Proposition 1

Theorem 1. Fix P, S_t, a, c, and an admissible replay specification ω under Assumption 1. The verifier checks the registered components, action type, branch universe, and current state. The trusted executor derives the result and permitted state changes from that verified state. The action modifies only the work covered by the join.

For MajorityJoin, the leader exceeds the runner-up by more than the unresolved vote budget. Assigning every remaining vote to an alternative cannot overturn the leader. The generalized condition in Eq. (6) gives the same result for class-specific residual bounds. The shortcut therefore preserves the winning answer.

For FirstSuccessJoin, the registered acceptance test accepts a completed witness. Continuing sibling branches cannot invalidate its existence. VerifierAcceptJoin follows the same argument for its registered proof predicate. Both preserve the existential declared output. If the declared output selects a particular witness, the certificate must also establish its registered selection rule, as specified in Sec. 2.

For AllRequiredJoin, the conditions in Tab. 2 apply to the post-action sets and retain every required branch in completed or deferred work. Deferred work completes under Assumption 1. A terminal action has already completed every required branch. Both actions preserve required-subtask completion.

In each case, the existing ordered effect trace is unchanged. Canceled branches contribute no external effects, and the remaining commits are determined by the preserved declared output and fixed input context. Hence the future effect traces are equal as well. If a commit depends on a candidate's identity or content, that information belongs to the declared output and its selection rule must be verified. Together these facts establish Eq. (4).

Proposition 1. Let e be the resolved subexpression and let σ assign the unfinished leaves values in their declared domains, consistently with S_t. Domain coverage includes every admissible completion, and every occurrence of a shared result uses the same value. By the certificate condition, $e(\sigma) = v$ for every such σ. Replacing e by v therefore leaves its parent operator's value unchanged. Repeating this argument along all paths to the output gives $E(\sigma) = E[e \leftarrow v](\sigma)$.

After substitution, cancel a call only if none of its unfinished results is needed, directly or through another call, by any remaining declared output. Such cancellation leaves every retained dependency available, including shared results used elsewhere. Later branch results restrict the admissible assignments, so earlier substitutions remain valid. Repeating the argument preserves the declared outputs. Assumption 1 preserves committed effects as in Theorem 1, so these local actions satisfy Definition 1. When the root is fixed, the same argument justifies the final return. The declared output can include a Boolean outcome, candidate identity, or ordered list.

A.2 Majority Bounds and Stopping Conditions

For answer class z, define the completed count and an admissible upper bound on unresolved votes as

$$N_t(z) = \sum_{b \in \mathcal{B}_t^{\text{done}}} \mathbf{1}[\text{canon}(y_b(t)) = z], \qquad R_t(z) := \sum_{b \in \mathcal{B}_t^{\text{live}}} u_b(z), \tag{5}$$

Each $u_b(z) \in \{0, 1\}$ is a sound possibility bound: for every admissible completion, a live branch that returns z has $u_b(z) = 1$. The actual additional votes $V_t(z; \omega)$ therefore satisfy $V_t(z; \omega) \le R_t(z)$ for every $\omega \in \Omega(P, S_t)$. These bounds use the branch contract and observed state. Setting $u_b(z) = 1$ for every live branch and admissible class gives $r_{\max} = |\mathcal{B}_t^{\text{live}}|$. With leader $z_t^\star = \arg\max_z N_t(z)$ under the declared tie rule, the certificate requires

$$N_t(z_t^\star) > \max_{z \neq z_t^\star} \big(N_t(z) + R_t(z)\big), \tag{6}$$

The branch-universe hash, canonicalizer, residual bound, and tie rule fix the admissible votes and their equivalence classes.

Proposition 2 (Certificate existence under strict-winner semantics). Consider N total branches with d completed. Let m_1 and m_2 be the leader and runner-up counts. If every unresolved branch may vote for any answer class and the join requires a strict winner, a valid MajorityCert exists if and only if $m_1 > m_2 + (N - d)$.

Proof. The inequality is sufficient by Eq. (6). If it fails, assigning all $N - d$ remaining votes to the runner-up permits a tie or reversal, so the current leader is not guaranteed to remain a strict winner. Under a tie rule that favors the leader, equality can preserve its result. The implemented strict-margin check rejects this equality case.

A.3 Examples of Certified Decisions

Figure A1 applies the certificate conditions to a vote, a tested program, required subtasks, and a registered witness predicate. The first-success and verifier-accept examples preserve the Boolean acceptance result of the registered predicate. If the rule selects the first success in a fixed priority order, the certificate must also show that no unfinished higher-priority branch can replace the candidate.

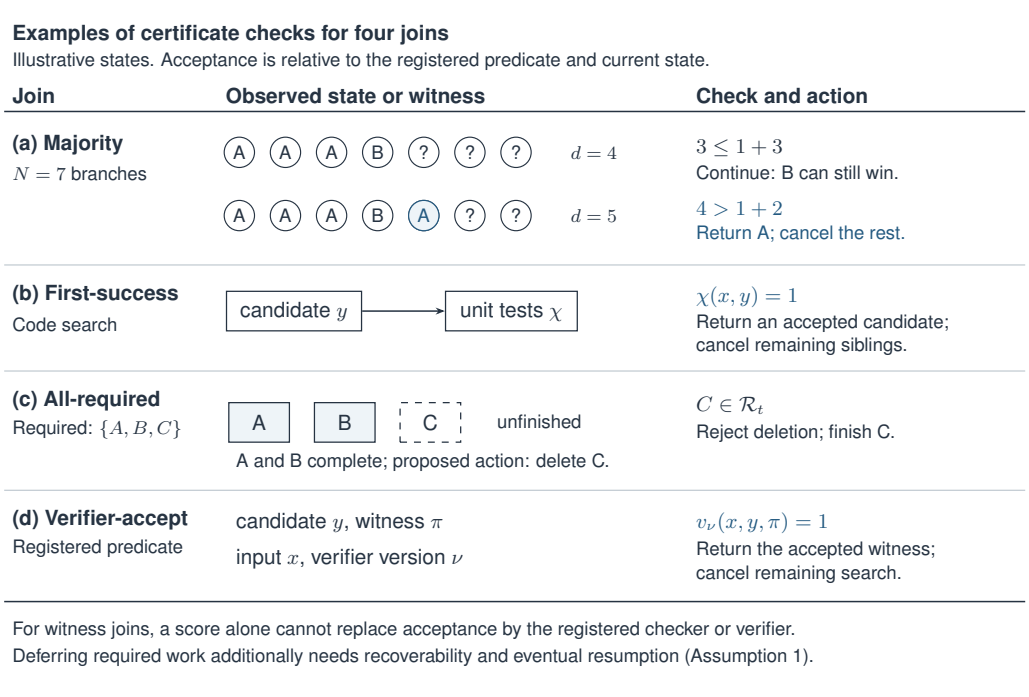

Figure A1: Illustrative checks for four joins. Majority uses completed counts and the unresolved vote budget. First-success requires an accepted test witness. All-required rejects deletion of unfinished required work. Verifier-accept checks a supplied witness against the registered predicate.

B Runtime Implementation

B.1 Certificate Schema and State Bindings

Figure A2 shows the completion loop, state bindings, verification, and execution actions. Table A1 lists each certificate's fields.

The verifier reconstructs state hashes and checks certificate metadata against registered components. It rejects missing registrations and mismatched bindings before running acceptance tests. With valid bindings, Majority verification recomputes vote counts and the residual budget. FirstSuccess confirms the candidate, input, witness, test version, and execution environment, then reruns the acceptance test. VerifierAccept evaluates its registered predicate on the current input and witness.

For a local certificate, the executor cancels calls whose results no remaining output needs, then continues. For a terminal certificate, it returns the verified result and cancels the covered remaining work. The registered output expression determines any candidate-selection rule. The completion loop serializes state updates, verification, and commit, so each action commits on the state version it was verified against.

We test these bindings with matching, changed, missing, and tampered values. An 800-program HumanEval/MBPP audit with execution-environment binding enabled reproduces the reported outcomes. HumanEval and MBPP run tests in temporary subprocesses and discard their state, so these programs commit no persistent external effects.

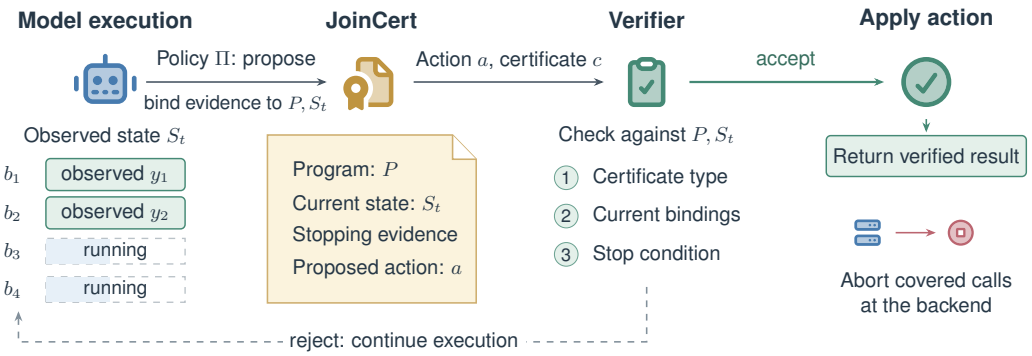

Figure A2: CertiJoin control flow for terminal joins. The policy proposes actions from the observed state. Accepted certificates authorize return and cancellation, while rejected proposals return to the completion loop. The audit log retains each decision and its evidence.

Table A1: State bindings and witness fields for deterministic certificates. The action type selects the join operation, whose executor derives the returned value and cancellation targets from the verified state. AllRequired sets describe the proposed post-action state.

Certificate	Bound state / witness fields	Verifier check
MajorityCert	Join id, branch-universe hash, canonicalizer id and version hash, tie rule, leader and runner-up answer hashes, m_1, m_2, and $r_{\max}$.	Recompute canonicalized vote counts from the current state, confirm the declared branch universe and tie rule, and check $m_1 > m_2 + r_{\max}$.
FirstSuccessCert	Candidate hash, acceptance-test id and version hash, input hash, execution-environment hash, witness hash, and pass bit.	Rerun or verify the registered deterministic acceptance test on the hashed input and witness, confirming that the declared first-success predicate is already satisfied.
AllRequiredCert	Required-set hash, completed-set hash, deferred-set hash, deleted-set hash, and state hash.	Check post-action coverage and non-deletion. A terminal action requires all required work completed. Deferral requires recovery and completion under Assumption 1.
VerifierAcceptCert	Candidate hash, witness-predicate id and version hash, proof or witness object, and state hash.	Check acceptance by the registered deterministic witness predicate.

B.2 Logical State and Backend Resources

A live branch can be represented as

$$s_b(t) = \big(n_b(t), y_b(t), \delta_b(t), \kappa_b\big), \tag{7}$$

where $n_b(t)$ points to backend execution state, $y_b(t)$ is its generated suffix, $\delta_b(t)$ is the remaining resource budget, and κ_b is the controller-declared continuation class. The verifier uses logical branch identities, outputs, and join bindings. The backend owns physical KV storage.

Shared-prefix and paged-KV backends retain common blocks across branches (Kwon et al., 2023; Zheng et al., 2024; Miao et al., 2023). Their conceptual accounting changes from per-branch storage to distinct blocks:

$$\mathrm{KV}_{\text{naive}}(t) = \sum_{b \in \mathcal{B}_t} k_b(t) \quad \text{to} \quad \mathrm{KV}_{\text{shared}}(t) = \sum_{n \in V_t} k_n(t), \tag{8}$$

Here V_t contains the distinct blocks used by live branches. The quantities $k_b(t)$ and $k_n(t)$ measure per-branch and per-block KV storage in bytes. Cancellation removes a branch's references, and shared blocks remain available to their other users (Fig. A3). Abort records and aggregate KV usage track the backend's response. Offload and cache placement control storage independently of these logical references (Sheng et al., 2023; Patel et al., 2024; Liu et al., 2025).

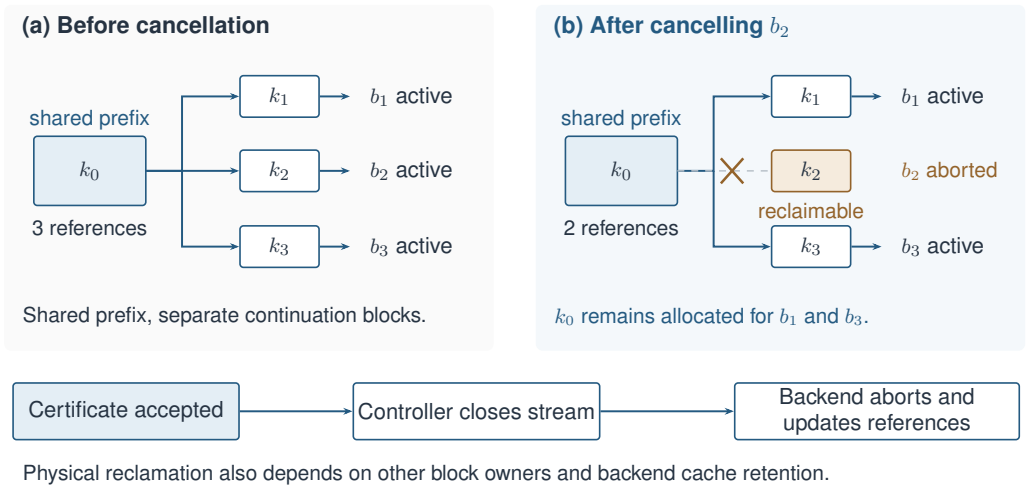

Figure A3: Certified cancellation above a shared-KV backend. Branches share prefix block k_0 and have separate continuation blocks. Canceling b_2 removes its references. Branches b_1 and b_3 still need k_0, while k_2 becomes reclaimable if no other owner or cache retains it. The backend determines when to release storage after the verified logical action.

C Experimental Protocols

C.1 Workloads and Stopping Policies

Table A2 lists the model, branch-budget, and replay settings. The experiment records include prompt hashes, acceptance-test versions, branch seeds, and proposal decisions so that paired methods use the same execution specification. Search and tool-using agents can combine branches at several points (Zhou et al., 2023; Yao et al., 2023b; Liu et al., 2024; Jimenez et al., 2024). For our search, debate, and code-validation replays, App. D.1 specifies which output each join must preserve. Execution comparisons also include committed external effects, as defined in Sec. 2.

The main majority traces use Qwen2.5-7B-Instruct, Qwen2.5-14B-Instruct, Qwen2.5-32B-Instruct, and Meta-Llama-3.1-8B-Instruct at temperature 0.7 and top-p 0.95. Generation limits are 512 tokens for GSM8K, 768 for MATH500, and 256 for MMLU. Canonicalizers first read the answer after a final `####` marker or inside `\boxed`. Otherwise, GSM8K uses the last number after removing commas, dollar signs, and trailing periods. MATH500 strips `\left`, `\right`, and spacing, falling back to the last nonempty line. MMLU maps an option letter or matching choice text to A–D and gives each unrecognized output its own invalid class. Ties are broken lexicographically, and fixed-trace replay processes branches in ascending index order. The released code and data list the identifiers, version hashes, and seeds of all 12 runs.

Table A2: Measurement conventions and experimental configuration summary. Full identifiers and hashes are included in the released artifacts.

Item	Value
Workloads and model scope	
Model backbones	Qwen2.5 7B/14B/32B and Llama3.1 8B
Branch counts	Majority sweep: $N \in \{4, 8, 16\}$, with $N = 16$ in the main slice. Certaindex: $N = 30$. Verifier microbenchmarks: $N \leq 32$
Datasets	GSM8K, MATH500, MMLU, HumanEval, MBPP, and ClassEval-derived code programs
Serving and method settings	
Serving replay	GSM8K, $N = 8$, at 30/60/120 QPM with 100 requests per policy
Streaming cancellation	Main Qwen2.5 7B crossover at 30 QPM, 200 request executions per policy. Additional settings in Table A15
Wave-based live serving	GSM8K with 200 requests per policy: Qwen2.5 7B at 30/60/120/240 QPM and Llama3.1 8B at 60 QPM
Compared methods	Full, Dedicated short circuit, direct margin checking, published stopping policies, heuristic controls, and CertiJoin
Accounting and audit trail	
Latency accounting	Recorded branch latency through the selected stop plus certificate construction and verification. Audit serialization measured separately
Acceptance-test accounting	HumanEval/MBPP include deterministic acceptance test re-execution time
Coverage accounting	Coverage is the fraction of proposed shortcut actions accepted
Logged artifacts	Branch seeds, prompt/template hashes, acceptance-test versions, proposal logs, and verifier decisions

The majority proposal policy, called Aggressive when its proposals are accepted without certification, proposes the current unique leader once its fraction of valid observed answers reaches τ, starting at the first completion. The $N/2$ variant waits for at least half the branches. Policy-triggered CertiJoin checks these proposals with MajorityCert and continues after rejection. Direct margin checking and every-completion certification are evaluated separately.

The unchecked code control returns the first candidate without testing it. Its full reference runs native test-and-stop execution until a candidate passes. Certification verifies the witness's state bindings and reruns its test. In the AllRequired workload, the verifier rejects every proposed deletion of required work, and all required branches complete.

The learned stopper is a gradient-boosted classifier that predicts whether the current unique leader matches the full-join answer. Its 13 features use only answers received so far: completed and remaining branch counts, leader and runner-up counts and fractions, their margin and residual-budget-adjusted margin, answer entropy, distinct-answer count, leader uniqueness, and the mean and standard deviation of observed branch token lengths. Hashing program identifiers assigns 60% to training and 20% each to calibration and test partitions, keeping every prefix and model of a question together. The classifier uses `GradientBoostingClassifier(random_state=17)` with default hyperparameters and labels from completed training traces. The sweep uses the training and test partitions, reporting every pre-specified threshold on the test set. A predicted probability above the threshold proposes a stop. Appendix D.9 gives the complete grid.

C.2 Replay Protocols and Metrics

Sequential replay sums recorded generation times through the selected stop and adds certificate costs. Paired serving replay records complete branch outputs and vLLM call latencies under the offered load, using exponential interarrival times with rate QPM/60 and default seed 9100. Policies share these recorded latencies, including queueing during trace collection. The wave and streaming experiments measure elapsed backend execution, where batching, prefill/decode scheduling, and cache placement affect latency (Yu et al., 2022; Agrawal et al., 2023; Zhong et al., 2024; Srivatsa et al., 2024). Each protocol specifies its timing interval and how the controller stops work.

Let $\mathrm{Outcome}(P, M; \omega)$ be the terminal declared output and ordered committed-effect trace for method M. The paired per-instance violation indicator is

$$\mathrm{Violation}(P, M; \omega) = \mathbb{1}\Big[\mathrm{Outcome}\big(P, M; \omega\big) \neq \mathrm{Outcome}\big(P, \mathrm{full}; \omega\big)\Big], \tag{9}$$

For witness joins, the declared output records whether a returned candidate passes the registered acceptance test. We compute this Boolean value for both methods. Invalid returns are counted separately when a returned candidate fails the test. Thus two unsuccessful runs can agree on the declared output even if one returns an invalid candidate. Table A9 reports both metrics on the same requests. Dataset means are

$$L(M) = \frac{1}{|\mathcal{D}|} \sum_{P \in \mathcal{D}} T(P, M), \qquad \mathrm{ViolRate}(M) = \frac{1}{|\mathcal{D}|} \sum_{P \in \mathcal{D}} \mathrm{Violation}(P, M; \omega_P), \tag{10}$$

where $T(P, M)$ includes certificate generation and verification. The paired specification ω_P is fixed across methods. For request i, relative saving is $s_i = 1 - T_i^{\mathrm{cert}}/T_i^{\mathrm{full}}$, and a request benefits when $s_i > 0$.

For an action proposed at logged state S_t, counterfactual replay checks whether applying the action would change the declared output or committed effects:

$$\mathrm{Unsafe}(P, S_t, a, \omega) = \mathbb{1}\Big[\mathrm{Exec}_{\mathcal{J}}(P, S_t; \omega) \neq \mathrm{Exec}_{\mathcal{J}}\big(P, a(S_t); \omega\big)\Big], \tag{11}$$

Let N_{prop}, N_{cert}, N_{unsafe}, and N_{rej} count proposed, certified, outcome-changing, and rejected outcome-changing actions. The last two counts use the complete replay as an oracle. Then

$$\mathrm{Coverage} = \frac{N_{\mathrm{cert}}}{N_{\mathrm{prop}}}, \qquad \mathrm{UnsafeReject} = \frac{N_{\mathrm{rej}}}{N_{\mathrm{unsafe}}}. \tag{12}$$

Coverage and rejection rates use action counts because a request may issue several proposals before a certified stop.

C.3 Stopping Rules and Completion Orders

Adaptive-Consistency. AC tests $\Pr[X > 1/2] \geq C$ with $X \sim \mathrm{Beta}(v_1 + 1, v_2 + 1)$, where v_1, v_2 are the two largest observed answer counts and $C = 0.95$ (Aggarwal et al., 2023). Each completion permits a proposal, which MajorityCert checks before authorizing a stop. Rejection keeps the outstanding branches running.

ESC. ESC checks non-overlapping complete windows within the $N = 16$ budget (Li et al., 2024). A unanimous window proposes its answer. We use window 5 for GSM8K and 8 for MATH500. After rejection, the next complete window can propose again. If none passes, execution completes all branches.

Certaindex. We implement the paper's self-consistency stopping rule (Fu et al., 2025) at the policy interface from its published description; the public Dynasor revision `d0bf9c38305d` releases the chain-of-thought probing server without this scheduler. Certaindex uses stability $1 - H/\log n$, where $H = -\sum_z p_z \log p_z$ and n is the number of sampled branches. One-shot checking occurs after five branches, and five-step checking repeats after each five-branch wave. The published appendix uses normalized-score threshold 0.7 on GSM8K. We also test normalized-score threshold 0.5 for sensitivity, which differs from the original paper's raw-entropy threshold 0.5.

These rules run on a trace of 6,000 branches from the first 200 official GSM8K test questions, generated with Llama3.1 8B at temperature 0.7, top-p 1.0, and a 512-token maximum. Generation uses two 64-GB Ascend 910 accelerators, tensor parallelism two, CANN 8.5, vLLM/vLLM-Ascend 0.13.0, PyTorch 2.8, and torch-npu 2.8.0.post2. The recorded GSM8K canonicalizer groups answers as exact strings, with lexicographic ties.

Completion orders. The random-order experiment applies 100 permutations to each of 800 fixed model-request branch multisets per dataset, giving 80,000 evaluations per policy. Seed 20260727 generates the shared permutations. The runner-up-first stress order places votes for the highest-count losing answer first and retains the relative order of all remaining branches. We construct this order offline from the full result, then reveal only the arriving prefix to each policy. Appendix D.5 reports both tests.

C.4 Live Serving and Reference Audit

The main live run uses Qwen2.5 7B with vLLM 0.13.0 on one accelerator, 64 active sequences, an 8,192-token batch limit, and 30 QPM. Sampling uses temperature 0.7, top-p 0.95, and at most 512 generated tokens per branch. Both policies use the same 100 GSM8K questions, seeded arrival schedule, and branch seed assignment. The Full, CertiJoin, CertiJoin, Full crossover produces 200 request executions per policy. Reversing run order reduces sensitivity to shared-system variation.

Latency runs from scheduled arrival to program commit, including controller and backend queues. Token counts include generation through cancellation. After each run, the audit regenerates missing branches from their original prompts and seeds and combines them with observed outputs. Concurrent scheduling can change outputs even with fixed seeds, so the audit provides a reconstructed full join. Fixed-trace comparisons use identical recorded branch outputs. In both protocols, the majority condition covers every admissible unfinished vote.

We count a cancellation as backend confirmed when the server has admitted the request and logged its matching abort. Focused probes verify that eight streams are admitted and running before closing them, then inspect abort records and aggregate KV usage. Their results are in App. D.8.

D Additional Results and Diagnostics

D.1 Reuse Across Programs

Each adapter fixes the output rule, branch identities, and result domains before receiving branch results. The shared core in Sec. 3.1 enumerates up to 100,000 assignments. The largest evaluated Boolean expression has 16 leaves, or 65,536 assignments, well within this budget; a larger input raises a budget error and receives no certificate, so no shortcut can be taken. One enumeration procedure handles finite conditionals, candidate selection, and votes. For monotone AND/OR expressions, evaluating all unknown leaves as false and then as true gives the same exact test in two evaluations, including shared leaves. Affine top-k scores use Eq. (2), which computes the exact minimum over the declared product domain and combines repeated uses of each leaf.

The ToT adapter uses published GPT-4 Game of 24 traces with four search levels (Yao et al., 2023a). Each level passes a ranked candidate list, or ordered beam, to the next. Candidate scores sum three parser ratings scaled by 1,000. Each rating belongs to the fixed domain $\{0, 1, 1000, 20000\}$, with zero for an unrecognized rating, so the score domain contains every possible sum before execution. Repeated uses of a cached scoring prompt share one leaf. The core preserves all 400 beams in Table 3.

The debate adapter checks final aggregation in Qwen2.5-7B-Instruct traces from the published three-agent, two-round protocol (Du et al., 2024b), covering 4 development and 20 evaluation questions. We test two output rules under two fixed completion orders. The reference implementation (revision `9846749350eb`) votes over parsed answers and requires the last agent's answer to be parseable. The second rule uses the vote alone. Early stopping begins at final aggregation, preserving the messages used by the second round. Three labels and an unparseable value encode known and unseen answers. Since aggregation depends on equality and parseability, this domain covers every partition of the three agents' answers before their responses arrive.

The code-validation adapter uses Qwen2.5-7B-Instruct traces from ClassEval-derived tasks (Du et al., 2024a), covering 4 development and 14 exploratory evaluation classes. Each bounded program has 8–12 component checks across 2–4 candidates, and a candidate qualifies only if all its components pass. Every unfinished check has the fixed domain $\{\text{pass}, \text{fail}\}$. We test three output rules under two fixed completion orders: whether a qualifying candidate exists, the lowest qualifying identifier, and the highest qualifying identifier.

Across 6,567 observed partial states, the shared core and independent dedicated checkers make identical stopping decisions. A separate exhaustive check covers 4,378 partial states from 17 small program variants, also with no differences from independent completion enumeration.

Stale hand-written stops. A hand-written stop fixes its output rule and branch budget in code, so it goes stale when either changes. For a rule change, we replay the dedicated code checker for the lowest qualifying identifier on programs that declare the highest, and the reverse, over all 18

classes and both completion orders. Of these 72 stale stops, 12 return a different candidate, all in the 3 classes where several candidates qualify; 5 classes have any qualifying candidate. For a budget change, we apply the strict-margin check written for 16 votes, $m_1 > m_2 + \max(0, 16 - t)$ after t completions, to the 200-question, 30-branch GSM8K trace of Sec. 4.4. It stops every run early and changes 5 answers. The verifier rejects every stale stop, because code certificates bind the declared selection rule and MajorityCert recomputes the unresolved votes from the bound 30-branch universe. Certified execution on the same trace ends 194 of the 200 runs early without changing an answer.

D.2 Nested Code Validation: Replay and Online Results

The serial replay fixes candidate and component identities before execution. A candidate qualifies when all its components pass. The first block of Table A3 follows the fixed order until the declared output is determined. The second also skips unfinished components of rejected candidates and preserves whether a qualifying candidate exists. Dedicated short circuit and the shared core execute the same components, whose recorded generation determines the token count. Selection rules change the required evidence: on the DataStatistics trace, the lowest qualifying identifier takes three checks and the highest takes twelve.

Table A3: Serial code-validation replay on 14 classes. Full execution uses 148 component checks and 56,036 tokens. Early counts classes finishing with fewer checks. Candidate order visits one candidate's components first, while component order visits each component across candidates.

Selection rule	**Order**	**Checks ↓**	**Early ↑**	**Tokens saved ↑**
Program termination only (prefix replay)				
Any qualifying candidate	Candidate	102/148	12/14	29.91%
Any qualifying candidate	Component	98/148	12/14	38.87%
Lowest identifier	Candidate	102/148	12/14	29.91%
Lowest identifier	Component	98/148	12/14	38.87%
Highest identifier	Candidate	133/148	8/14	12.34%
Highest identifier	Component	108/148	8/14	31.55%
Candidate pruning and program termination				
Any qualifying candidate	Candidate	71/148	13/14	53.25%
Any qualifying candidate	Component	90/148	13/14	43.87%

The online experiment uses the same 14 exploratory classes and all their components, with two to four candidates and 8 to 12 model calls per program. Each candidate must pass its component tests and the original full-class integration suite. The declared output records whether such a candidate exists. The adapter assigns each check the domain $\{\text{false}, \text{true}\}$ and combines candidate checks through the core's conditional operator. Candidate certificates authorize local cancellation, and root certificates authorize return. Dedicated short circuit evaluates these conditions directly.

Each method admits jobs in candidate order with at most four generation/check jobs active. Full execution completes the declared model-call budget. Both stopping methods skip pending jobs and abort running generation after a candidate rejection or root decision. Generated Python executes in a dedicated chroot with dropped privileges, resource limits, and a system-call filter. Prompts contain class specifications and the supplied constructor. Reference method implementations do not enter prompts.

Two independent backends each serve Qwen2.5-7B-Instruct in BF16 on one 64-GB Ascend 910 device, using SGLang 0.5.18 with per-request seed support, tensor parallelism one, an 8,192-token context, and radix caching disabled. Each backend runs one program at a time, and all methods for a class use that same backend. Sampling uses temperature 0.7 and top-p 0.9. Component budgets are $\min(2048, \max(768, 384m))$ tokens for m requested methods. Two repetitions use branch seeds derived from class, candidate, component, and repetition identifiers. Method order rotates by class and reverses in the second repetition. A development smoke test and seed probes precede evaluation and are excluded from Table A4.

Latency runs from controller entry to the final program decision, including generation and test execution. Token totals include every admitted stream through its final backend frame, including aborted streams. The log records the decision, abort request, acknowledgement, and stream drain. Controller check time includes monitor construction, observed-state updates, stopping checks, certificate construction, and verification. The shared core averages 18.50 ms per program, with p95 44.55 ms. Table A4 lists the totals behind Figure 3.

Table A4: Online nested code validation with Qwen2.5 7B. Each method runs twice on each of 14 classes. Tokens and started calls are totals. Decision time is the mean per run for the stopping methods, including certificate construction and verification for CertiJoin.

Method	**Mean latency** (s) ↓	**Tokens** ↓	**Started** calls ↓	**Decision time** (ms) ↓
Full	34.54	108,638	296	—
Dedicated	21.49	73,273	280	0.07
CertiJoin	21.55	73,247	281	18.50

We estimate uncertainty with 10,000 bootstrap draws over the 14 classes, keeping methods and repetitions paired. The 95% intervals for mean-latency and token reductions relative to Full are 27.3 to 47.8% and 22.3 to 42.9%. Relative to Dedicated, the signed changes are $+0.27\%$ for latency (95% interval: -0.62 to $+0.94\%$) and -0.04% for tokens (-0.26 to $+0.10\%$), with positive values indicating increases.

Each method finds an accepted candidate in 1 of 28 runs. Among the 125 component calls completed by both CertiJoin and paired Full execution, 115 have identical text and 124 have identical test outcomes.

The decision audit reproduces 359 online stopping decisions and verifies 375 certificates. The shared and dedicated checkers also agree at all 297 prefixes of the complete runs and all 729 states in an independent exhaustive test with explicit integration leaves.

D.3 Direct Stopping and Verification Cost

The direct baseline evaluates Eq. (3) after every completed branch. The second method constructs and verifies a MajorityCert at the same points. The third verifies proposals from the policy in Sec. 4.1. All three use the same 32,000 recorded branches from the 2,000-request slice, in the same completion order.

Table A5: Mean completed branches on the fixed majority slice. The two CertiJoin columns differ in when verification is invoked. All three methods preserve every full-run answer.

			CertiJoin	
Dataset	**Requests**	**Direct**	Every completion	Policy proposals
GSM8K	800	9.506	9.506	9.523
MATH500	800	12.159	12.159	12.349
MMLU	400	9.438	9.438	9.443

Direct checking and certification after every completion select identical stopping positions and generation prefixes on all requests. Checking every completion yields 1,800 early exits, compared with 1,690 when verification follows policy proposals. Proposal timing accounts for 115 later stopping positions. The main replay tables use the latter configuration.

Controller timings cover state updates and checks over a complete request on an aarch64 CPU with Python 3.11.15. Mean controller time is 0.023 ms for direct checking, 1.355 ms for certification after every completion, and 1.063 ms for certification of policy proposals. Audit serialization is measured separately, averaging 0.050, 0.055, and 0.055 ms, respectively.

Verification cost depends on the join. Reconstructing state bindings and majority or all-required summaries takes $O(N)$ time. First-success and verifier-accept checks add $O(T_\chi)$, where T_χ is the

acceptance-test runtime. In verifier microbenchmarks at $N \leq 32$, p95 majority verification is 0.11 to 1.15 ms. Code acceptance tests take 51 to 63 ms on average. Appendix D.9 reports how often proposals are accepted, together with budget sensitivity and verifier tests.

D.4 Paired Majority and Code Results

Table A6: Declared-outcome disagreement across backbones. Majority uses policy-triggered verification with $N = 16$, $\tau = 0.5$. Witness rows compare unchecked shortcuts with certified returns. Speedup includes certificate overhead. Aggressive accepts the $\tau = 0.5$ proposals without certification. K counts programs per model or requests per policy and load. A dash marks an unevaluated $N/2$ baseline.

		Full-join disagreement ↓			Efficiency
Workload	K	**Aggressive**	$N/2$	**CertiJoin**	**CertiJoin speedup**
Majority joins					
GSM8K	200	1.5–10%	0–1.5%	**0%**	1.53–1.73×
MATH500	200	26–54%	1–2.5%	**0%**	1.08–1.28×
MMLU	100	0–15%	0–3%	**0%**	1.55–1.75×
First-success joins					
HumanEval	100	6–17%	—	**0%**	0.98–0.99×
MBPP	100	4–14%	—	**0%**	0.99–1.00×
Paired serving replay (30/60/120 QPM)					
GSM8K	100	1–16%	—	**0%**	1.37–1.54×

Table A7: Answer changes on 2,000 paired requests, including wrong-to-wrong changes. C→W and W→C denote changes in benchmark correctness, as percentages of all requests. Full accuracy is 74.65%. Δ accuracy is measured in percentage points.

Policy	**Changed answers** ↓	**C→W** ↓	**W→C**	**Δ accuracy** ↑
Aggressive	16.60%	5.70%	1.45%	−4.25
$N/2$	1.15%	0.40%	0.20%	−0.20
CertiJoin	**0%**	**0%**	0%	**0**

The fixed-slice analysis matches 2,000 model-request pairs by request and replay identifiers. Full execution has 1,493 correct answers, aggressive stopping 1,408, the $N/2$ variant 1,489, and CertiJoin 1,493. Aggressive stopping changes 332 answers: 114 correct-to-wrong, 29 wrong-to-correct, and 189 wrong-to-wrong. The $N/2$ variant changes 23. Per-request relative savings have a median of 42.154% and a 95th percentile of 45.994%. Table A8 gives model-level results.

D.5 Completion-Order Sensitivity

Certified execution preserves every full join under random permutations and the runner-up-first order. The 100 permutations per request vary completion order within a fixed branch multiset, while the runner-up-first order concentrates misleading early votes. AC changes 13 of 80,000 GSM8K outcomes and 88 of 80,000 MATH500 outcomes under random permutations, while ESC changes 5 and 0, respectively. Tables A11 and A12 report the costs and disagreement rates. Table A10 gives the recorded-order speedups and the stress-order disagreement shown in Figure 1(c).

D.6 Speed-Matched Stopping Rules

We match each rule family to CertiJoin's recorded-order replay cost on the 2,000-request slice. From each grid, we select the fastest setting no faster than certification after every completion (App. D.3). The grids cover every fixed budget, window, and vote margin up to 16, and AC thresholds from 0.95 to 0.9999. We then evaluate the selected settings unchanged under the runner-up-first order defined in App. C.3.

Table A8: Per-model results on the fixed $N = 16$, $\tau = 0.5$ majority slice. The first three numeric columns report answer disagreement percentages. Speedup uses sequential replay time including certificate costs.

Dataset	Model	Aggressive	$N/2$	CertiJoin	Speedup
GSM8K	Qwen2.5 7B	6.50%	1.50%	0.00%	1.661×
GSM8K	Qwen2.5 14B	3.50%	0.50%	0.00%	1.683×
GSM8K	Qwen2.5 32B	1.50%	0.00%	0.00%	1.735×
GSM8K	Llama3.1 8B	10.00%	0.50%	0.00%	1.532×
MATH500	Qwen2.5 7B	27.50%	1.50%	0.00%	1.265×
MATH500	Qwen2.5 14B	26.00%	1.00%	0.00%	1.276×
MATH500	Qwen2.5 32B	28.50%	2.50%	0.00%	1.276×
MATH500	Llama3.1 8B	54.00%	2.50%	0.00%	1.076×
MMLU	Qwen2.5 7B	1.00%	0.00%	0.00%	1.725×
MMLU	Qwen2.5 14B	0.00%	0.00%	0.00%	1.741×
MMLU	Qwen2.5 32B	1.00%	0.00%	0.00%	1.746×
MMLU	Llama3.1 8B	15.00%	3.00%	0.00%	1.549×

Table A9: Witness outcomes on the same 100 programs per row. Full and unchecked success are counts of accepted witnesses. Disagr. is Boolean success disagreement (%). Invalid is the invalid-candidate return rate (%). Cert. D / I gives both rates for CertiJoin. Both code benchmarks use four models.

Task	Model	Full success	Unchecked success	Disagr.	Invalid	Cert. D / I
HumanEval	Qwen2.5 7B	94	88	6	12	0 / 0
HumanEval	Qwen2.5 14B	93	86	7	14	0 / 0
HumanEval	Qwen2.5 32B	94	87	7	13	0 / 0
HumanEval	Llama3.1 8B	86	69	17	31	0 / 0
MBPP	Qwen2.5 7B	81	71	10	29	0 / 0
MBPP	Qwen2.5 14B	88	84	4	16	0 / 0
MBPP	Qwen2.5 32B	87	83	4	17	0 / 0
MBPP	Llama3.1 8B	79	65	14	35	0 / 0

Table A10: AC and ESC on 800 model-request pairs per dataset, $N = 16$. Speedups use recorded-order sequential replay and include certificate costs. Disagreement with the full join is reported under recorded order (Rec.) and runner-up-first order (Stress); certified disagreement is zero in both.

		Disagreement (%) ↓			**Replay speedup** ↑	
Policy	**Dataset**	Raw/Rec.	Raw/Stress	Certified	Raw	Certified
AC (0.95)	GSM8K	0	4.13	**0**	3.160×	1.658×
AC (0.95)	MATH500	0	21.13	**0**	1.405×	1.202×
ESC ($w = 5$)	GSM8K	0	1.75	**0**	2.449×	1.461×
ESC ($w = 8$)	MATH500	0	0.13	**0**	1.113×	1.000×

Table A11: AC and ESC under 100 random completion-order permutations per request, $N = 16$ and 800 model-request pairs per dataset. Recorded-order results are in Table A10. Disagreement is measured against the full join. Certified execution checks each proposed stop with MajorityCert. Speedups include certificate costs and use sequential trace replay.

		Disagreement ↓		**Replay speedup** ↑	
Policy	**Dataset**	Raw	Certified	Raw	Certified
100 random permutations per request					
AC (0.95)	GSM8K	0.016%	**0%**	3.130×	1.655×
AC (0.95)	MATH500	0.110%	**0%**	1.411×	1.202×
ESC ($w = 5$)	GSM8K	0.006%	**0%**	2.464×	1.459×
ESC ($w = 8$)	MATH500	0%	**0%**	1.111×	1.000×

Table A12: Published policies under a runner-up-first order. Disagreement compares the returned answer with the full join over the same branches. Branch counts are for certified execution. Certaindex uses repeated five-step checks.

			Disagreement	**Disagreement**	
Policy	**Dataset**	**Requests**	Raw ↓	Certified ↓	**Branches**
AC (0.95)	GSM8K	800	4.13%	**0%**	9.73/16
AC (0.95)	MATH500	800	21.13%	**0%**	13.05/16
ESC ($w = 5$)	GSM8K	800	1.75%	**0%**	10.78/16
ESC ($w = 8$)	MATH500	800	0.13%	**0%**	16.00/16
Certaindex (0.7)	GSM8K	200	9.50%	**0%**	21.88/30

The selected fixed budgets change 74 of 2,000 recorded-order answers, including 63 on MATH500. The adaptive rules preserve all recorded-order answers and change up to 5.13% on MATH500 under the runner-up-first order. CertiJoin preserves every answer in both orders (Table A13).

D.7 Certaindex Thresholds and Retry Schedules

At threshold 0.7 on the $N = 30$ trace, raw and certified Certaindex return identical answers, and repeated five-step proposals retain 25.38% token savings under certification. A strict winner among 30 votes needs at least 16 completed votes, so branch 20 is the first five-step checkpoint that can be certified. The same margin arithmetic explains ESC on MATH500, whose first eight-answer window leaves eight votes unresolved. At threshold 0.5, certification removes both answer changes and restores 94% accuracy (Table A14).

The two disagreements at normalized-score threshold 0.5 are both correct-to-wrong. One five-branch prefix favors 140 by 3:2, although the full vote and gold answer are 20. The other favors canonicalized output `48g` over `48`, showing the role of answer normalization in the declared join. Both prefixes have stability 0.581834. Their 25 remaining branches can change the result, and certification rejects the proposals.

Table A13: Stopping rules matched to CertiJoin's recorded-order replay cost on 2,000 requests. Each rule uses the fastest grid setting no faster than CertiJoin in that order. Branches is the mean completed count. Changed answers are measured against the full 16-branch join in recorded and runner-up-first order.

					Changed (%) ↓	
Dataset	**Rule**	**Setting**	**Speedup ↑**	**Branches**	**Recorded**	**Stress**
GSM8K	**CertiJoin**	—	1.673×	**9.51**	**0**	**0**
	Fixed budget	10 of 16	1.601×	10.00	1.00	2.50
	AC	$C = 0.999$	1.570×	10.12	0	0
	ESC	$w = 9$	1.546×	10.29	0	0
	Answer stability	$w = 9$	1.568×	10.14	0	0
	Vote margin	$k = 9$	1.605×	9.88	0	0
MATH500	**CertiJoin**	—	1.253×	**12.16**	**0**	**0**
	Fixed budget	13 of 16	1.230×	13.00	7.88	15.50
	AC	$C = 0.99$	1.242×	12.01	0	5.13
	ESC	$w = 6$	1.202×	12.46	0	5.13
	Answer stability	$w = 6$	1.239×	11.98	0	5.13
	Vote margin	$k = 7$	1.245×	12.01	0	1.75
MMLU	**CertiJoin**	—	1.698×	**9.44**	**0**	**0**
	Fixed budget	10 of 16	1.595×	10.00	0.75	3.75
	AC	$C = 0.999$	1.627×	9.91	0	0
	ESC	$w = 9$	1.623×	9.95	0	0
	Answer stability	$w = 9$	1.628×	9.90	0	0
	Vote margin	$k = 9$	1.654×	9.74	0	0

Table A14: Certaindex on the same 200 recorded-order requests, $N = 30$. Branches and token savings measure replay computation over the processed prefix. Full accuracy is 94%.

Threshold	**Policy**	**Disagr. ↓**	**Accuracy ↑**	**Branches**	**Tokens saved ↑**
0.7	One-shot	0%	94%	12.125	53.46%
0.7	One-shot + CertiJoin	0%	94%	30.000	0%
0.7	Five-step	0%	94%	9.350	63.75%
0.7	Five-step + CertiJoin	0%	94%	21.725	25.38%
0.5	One-shot	1%	93%	8.000	68.95%
0.5	One-shot + CertiJoin	**0%**	**94%**	30.000	0%
0.5	Five-step	1%	93%	7.025	73.03%
0.5	Five-step + CertiJoin	**0%**	**94%**	21.175	27.90%

D.8 Live Serving and Cancellation Records

Table A15: Additional live streaming experiments, 100 requests per policy and setting. Mean, p95, p99, and token columns give reductions (%) relative to full execution. Changed counts answers that differ from the reconstructed full join. Confirmed aborts require both a matching server admission record and abort record.

	Reduction (%) ↑					
Model / sequences / QPM	Mean	p95	p99	Tokens	**Aborts**	**Changed ↓**
Qwen2.5 7B / 64 / 60	13.0%	14.5%	13.0%	4.99%	653/653	**0/100**
Qwen2.5 7B / 16 / 30	9.1%	9.4%	9.4%	6.24%	648/648	**0/100**
Llama3.1 8B / 64 / 30	25.8%	25.5%	17.8%	6.14%	583/583	**0/100**

In the main run, mean controller queueing falls from 2.72 to 0.30 seconds and backend queueing from 16.86 to 13.48 seconds, adding to the latency reduction. Generation totals 805,606 tokens under Full and 763,084 under CertiJoin. The two main crossover repetitions reduce p95 latency by 23.7% and 25.7%, and p99 by 23.9% and 24.8%. Table A15 reports the other load, concurrency, and model settings. Every evaluated setting preserves all reconstructed full joins.

All 1,304 admitted cancellations in the main run have matching abort records. Each eight-stream probe produces eight matching aborts. At the next 250-ms poll, about 0.265 seconds after cancellation, the server reports zero running requests and zero aggregate KV allocation.

Wave-based serving. The wave controller stops dispatching later waves after a certified join. With 200 requests per policy for Qwen2.5 7B at 30, 60, 120, and 240 QPM and Llama3.1 8B at 60 QPM, it gives 1.09–1.15× mean speedups with zero join disagreements (Table A16). Streaming cancellation, which also aborts running requests, gives the larger gains reported above.

Table A16: Wave-based serving with four branches per wave and 200 requests per policy. Full and certified columns give mean latency. Disagreement uses each certified run's post-commit backfill reference. Negative p95 reduction denotes an increase.

Model	QPM	Full (s)	Cert. (s)	Speedup	p95 red. (%)	Disagr. (%)
Qwen2.5 7B	30	90.99	83.43	1.091	-0.84	0
Qwen2.5 7B	60	120.88	111.07	1.088	-0.57	0
Qwen2.5 7B	120	179.63	164.30	1.093	-0.44	0
Qwen2.5 7B	240	236.89	217.65	1.088	-0.42	0
Llama3.1 8B	60	134.28	117.22	1.146	0.15	0

D.9 Coverage, Budget Sensitivity, and Verifier Audits

Action coverage is the fraction of proposals accepted. Each request can issue several proposals before it stops. Table A17 reports coverage for the workloads in Table A6. Table A18 aggregates the larger proposal population across all majority backbones and (N, τ) settings. Every oracle-unsafe proposal is rejected.

Adaptive-policy grid. Table A19 reports all seven families: observed-fraction stopping ($\tau = 0.5$), fixed budget fractions $\{0.25, 0.5, 0.75\}$, answer-stability windows $\{2, 3, 4, 5\}$, vote margins $\{1, 2, 3, 4, 5\}$, Bayesian error thresholds $\{0.2, 0.1, 0.05, 0.01, 0.005\}$, sequential confidence levels $\{0.90, 0.95, 0.99, 0.995\}$, and learned probability thresholds $\{0.80, 0.90, 0.95, 0.99, 0.995\}$. Bayesian estimates use 512 Monte Carlo samples. Every policy has $N = 16$ and is evaluated on the same test questions within a dataset and model.

The released data contain every measured parameter, sample count, speedup, and disagreement rate. For each family and dataset–model pair, the data also record the test-grid point closest in speedup to CertiJoin, yielding 84 descriptive comparisons. Each records the threshold, both speedups, and their difference. Table A19 retains the full grid, and App. D.6 matches recorded-order cost on the complete 2,000-request slice.

Budgets and verifier audits. Table A20 gives early-stop and token-saving rates for every registered (N, τ) setting, aggregated over four backbones. Figures A4 and A5 show the associated proposal decisions and a recorded request. All 97,458 rejected majority proposals in these source runs fail the strict-margin condition.

The random-state audit draws N uniformly from 1–12, then a completed count uniformly from 0–N, a random completed-branch subset, and answers from $\{\mathrm{A}, \ldots, \mathrm{E}, \perp\}$, using seed 20260428. The reference implementation evaluates the strict-margin predicate independently of certificate construction. Across 100,000 states, 13,773 certificates are accepted and 86,227 rejected, with no decision mismatches. This checks the certificate implementation against the predicate proved in App. A.2.

Table A17: Proposal coverage for the workload settings in Table A6. Ranges span backbones and replay/load settings. All oracle-unsafe proposals are rejected.

Join / setting	Workload	Coverage	Unsafe rejection
Majority	GSM8K	10.5–11.0%	100%
Majority	MATH500	7.9–9.7%	100%
Majority	MMLU	9.9–11.1%	100%
First-success	HumanEval	36.9–58.4%	100%
First-success	MBPP	27.6–45.1%	100%
Serving replay	GSM8K	18.7–19.8%	100%

Table A18: Aggregate majority actions from the same 12 source runs as the main slice, across $N \in \{4, 8, 16\}$ and $\tau \in \{0.5, 0.6, 0.7, 0.8\}$. Every outcome-changing shortcut is rejected.

Workload	Proposed	Certified	Unsafe	Coverage	Unsafe rej.
GSM8K	52,726	9,081	588	17.2%	100%
MATH500	37,055	5,427	3,831	14.6%	100%
MMLU	26,821	4,636	297	17.3%	100%

Table A19: Adaptive-policy test grids, four backbones and $N = 16$. Each model has 44 GSM8K, 40 MATH500, or 15 MMLU test questions. Ranges span models and the stated parameter grids. The speedup and disagreement extrema can come from different grid points. CertiJoin uses the same questions.

Dataset	Policy family	Grid points	Speedup range	Disagr. (%)
GSM8K	Observed fraction	4	15.84–16.69×	2.27–4.55
GSM8K	Fixed fraction	12	1.33–4.10×	0.00–4.55
GSM8K	Answer stability	16	3.00–8.11×	0.00–4.55
GSM8K	Vote margin	20	2.43–16.69×	0.00–4.55
GSM8K	Bayesian	20	2.84–16.69×	0.00–4.55
GSM8K	Sequential CI	16	4.15–8.11×	0.00–2.27
GSM8K	Learned	20	7.01–16.05×	0.00–4.55
GSM8K	CertiJoin	4	1.59–1.75×	0.00
MATH500	Observed fraction	4	15.61–16.57×	22.50–65.00
MATH500	Fixed fraction	12	1.32–4.01×	2.50–50.00
MATH500	Answer stability	16	1.36–4.13×	5.00–37.50
MATH500	Vote margin	20	1.06–16.57×	0.00–65.00
MATH500	Bayesian	20	1.10–15.17×	0.00–62.50
MATH500	Sequential CI	16	1.08–1.96×	0.00–5.00
MATH500	Learned	20	1.05–4.96×	0.00–37.50
MATH500	CertiJoin	4	1.06–1.27×	0.00
MMLU	Observed fraction	4	15.41–15.94×	0.00–13.33
MMLU	Fixed fraction	12	1.32–4.12×	0.00
MMLU	Answer stability	16	3.01–7.97×	0.00
MMLU	Vote margin	20	2.46–15.94×	0.00–13.33
MMLU	Bayesian	20	2.63–15.94×	0.00–13.33
MMLU	Sequential CI	16	3.60–7.97×	0.00
MMLU	Learned	20	1.00–15.91×	0.00–13.33
MMLU	CertiJoin	4	1.66–1.77×	0.00

Table A20: Budget sensitivity on the main source runs. Stop columns report requests ending before N completions (%). The token columns report savings against the same full prefixes (%). Each setting contains 800 GSM8K, 800 MATH500, and 400 MMLU model-request pairs.

		GSM8K (%)		MATH500 (%)		MMLU (%)	
N	τ	**Stop**	**Tokens**	**Stop**	**Tokens**	**Stop**	**Tokens**
4	0.5	89.0	21.7	42.1	8.0	92.8	23.5
4	0.6	89.0	21.7	42.1	8.0	92.8	23.5
4	0.7	89.0	21.7	42.1	8.0	92.8	23.5
4	0.8	89.0	21.7	42.1	8.0	92.8	23.5
8	0.5	96.5	33.6	60.5	14.5	97.5	35.5
8	0.6	95.4	33.4	56.5	13.9	96.8	35.5
8	0.7	94.8	33.0	53.9	12.5	96.3	35.3
8	0.8	91.5	32.5	45.1	11.5	94.0	35.1
16	0.5	97.3	39.5	64.9	17.8	98.3	41.4
16	0.6	95.8	39.3	58.1	16.4	97.3	41.3
16	0.7	93.9	38.7	51.0	14.9	95.3	41.0
16	0.8	91.3	37.7	46.6	13.6	93.3	40.3

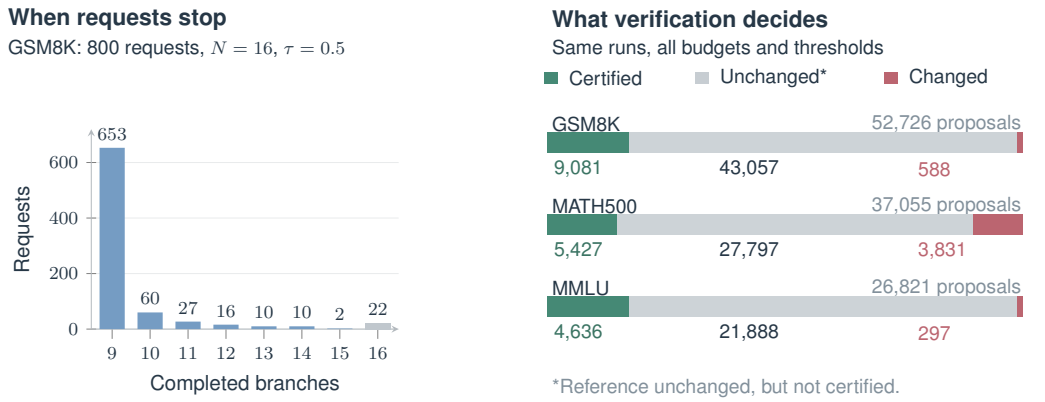

Figure A4: Verification decisions in the main source runs. Left: stopping positions for 800 GSM8K requests ($N = 16$, $\tau = 0.5$). Right: proposal outcomes across registered settings (Table A18). Unchanged denotes rejected proposals that match full execution.

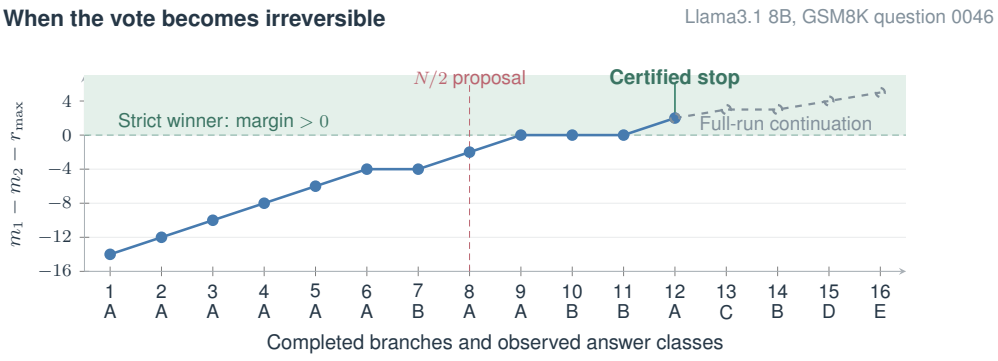

Figure A5: Recorded GSM8K request `gsm8k_test_0046` (Llama3.1 8B, $N = 16$). At branch 8, where the $N/2$ heuristic stops, the remaining eight votes could still elect B. CertiJoin stops at branch 12, the first point where A is guaranteed to win. Letters denote observed answer classes, and the dashed segment shows the remaining full-run trace.