Inductive Deductive Synthesis: Co-Evolving Code-and-Proof Generation

Shubham Agarwal[*,1], **Alexander Krentsel**[*,1], **Shu Liu**[*,1], **Mert Cemri**[*,1], **Audrey Cheng**[1], **Rui Meng**[2], **Tomas Pfister**[2], **Chun-Liang Li**[2], **Sylvia Ratnasamy**[1], **Aditya Parameswaran**[1], **Matei Zaharia**[1], **Ion Stoica**[1], **Mohsen Lesani**[3]

[1]UC Berkeley [2]Google [3]UC Santa Cruz

Abstract

AI agents increasingly excel at generating, testing, and refining code. However, they fall short on tasks requiring formal guarantees of full coverage that testing alone cannot provide. Distributed systems are a prime example: properties such as consistency between reads and writes must hold under every possible interleaving of events. Mechanized formal verification can guarantee such correctness, but typically demands months to years of expert effort. As evidence, even SOTA coding agents (Codex with GPT-5.4 and Claude Code with Opus 4.6) succeed on only 2/7 distributed key-value-store specifications. In this paper, we present the first effective approach to addressing this gap, Inductive Deductive Synthesis (IDS), which incrementally co-evolves implementation and proof, and learns from failed attempts to systematically try promising strategies. Built as an agentic LLM system, IDS achieves 7/7 in about 6.9 hours and $106 per spec on average, roughly 200× faster than expert effort and 17% cheaper than SOTA agents. IDS further incorporates performance feedback into the same loop, yielding implementations up to 3× faster than published verified systems.[†]

1 Introduction

Coding agents have made great strides at writing code for many tasks, even running tests to automatically refine their own output [5, 4, 41, 70, 50]. However, they fall short on tasks requiring correctness *guarantees* across every possible behavior, rather than just the cases reached during testing. Distributed systems are a prime example: properties such as consistency between subsequent writes and reads must hold under every possible interleaving of messages, failures, and concurrent updates, a combinatorially large space that no realistic test suite can cover [28]. The consequences of incorrect behavior are severe, ranging from a replicated store that loses data [6], to a file system that drops a write [43], to a confidential store that leaks a key.

To prevent such catastrophic failures, formal verification techniques can provide exhaustive correctness guarantees. These techniques allow a developer to (1) state a specification, (2) write an implementation, and (3) develop a machine-checkable proof that the implementation satisfies the specification on every possible input [9, 14, 26, 29]. However, adoption of formal methods over the past two decades has been limited due to the immense manual labor required: verifying real-world systems takes months to years of expert effort [24, 30, 11, 18, 65, 31].

Naturally, one may ask, can we simply provide a coding agent with a specification and formal verification tooling, and ask it to build complex systems that provably satisfy that specification? Our experiments demonstrate that the answer is no. We show in §5 that Codex (GPT-5.4) [41] and Claude Code (Opus 4.6) [4] fail to generate correct systems for 5 of 7 widely studied distributed

[*]Equal contribution. [†]Code available at https://github.com/skydiscover-ai/skydiscover.

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

key-value-store consistency specifications (e.g., causal consistency, monotonic reads). Following typical human patterns, agents treat verification as a downstream check on code they have already written. This approach (a) requires writing the whole proof at once, which is exceptionally difficult even for humans[1], and (b) defers all correctness feedback to the end of an implementation, depriving the agent of valuable guiding signals. If the implementation is wrong, no amount of proof effort can succeed; repairing the full implementation after proof failure is only partly helpful, as agents often lack the context to tell which design choices to rule out.

To address these challenges, we introduce *Inductive Deductive Synthesis* (IDS), a technique that enables agents to synthesize provably correct complex systems from a specification. Our key insight is to synthesize both the implementation and its proof *jointly and incrementally* (Fig. 1), while learning from failures at each step to guide the search toward promising solutions and away from dead ends. As detailed in §3, every implementation decision (e.g., adding a data structure or modifying control flow) is paired with a co-evolving proof update (e.g., adding new constraints or splitting lemmas).

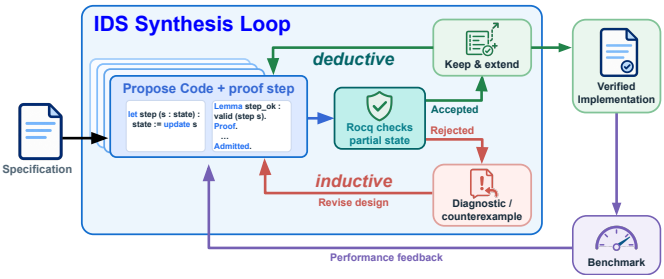


Figure 1: Code and proof advance jointly and incrementally with IDS; the proof assistant grades each partial step.

IDS offers two benefits: (1) each individual joint step is considerably simpler to *prove*, decreasing the complexity of proof generation, and (2) the intermediate *partial* proofs can be verified with a formal checker to assess the viability of the current design. A negative result rules out a dead-end implementation path before further work is wasted and serves as a counterexample to improve the design going forward. This is akin to "chain-of-thought" [63], except that the intermediate states are formally encoded and verified. Beyond correctness, this iterative co-evolution loop also optimizes performance: as soon as a candidate's implementation is complete, it is benchmarked on a distributed testbed; the performance measurements then further guide the search toward efficient implementations.

We instantiate IDS as a multi-agent system (§4), and we evaluate IDS on 7 distributed key-value-store consistency specifications: Chapar's published causal-consistency spec [31] plus six new *IDS suite* specs we release with the system (App. I). IDS produces a complete verified implementation on all 7 in about 6.9 hours and $106 per spec on average, with no human intervention or fine-tuning. Writing the six new specifications took an expert a few days (§6); the cost of a verified system thus moves from months of proof effort to a few days of specification plus hours of synthesis. On the two specs our vanilla agent baseline completed, IDS runs $1.6\times$ faster and 17% cheaper. The discovered implementations match or beat hand-written expert references on every spec and reach up to $3\times$ the throughput of Chapar's published vector-clock reference, a margin we attribute to a design-space search broader than what manual proof allows.

As a general prover, IDS achieves state-of-the-art performance on four cross-language verification benchmarks (DafnyBench [35], miniCodeProps [34], Verus-Bench [68], CoqStoq [56]), beating agentic baselines that share its prompt and underlying model by $+10\%$ to $+51\%$. It also leads on three proof benchmarks from real projects, including the Linux kernel: $39/60$ on VerKer [58] (prior $18/60$), $432/500$ on VeriSoftBench [67] (prior $205/500$), and $838/849$ on VeruSAGE-Bench [69] (prior $797/849$). On three code-and-proof synthesis benchmarks, IDS leads: $176/189$ on VERINA [73] (prior $38/189$), $65/77$ on AlgoVeri [74] (prior $31/77$), and a perfect $62/62$ on CloverBench [53].

In summary, we make the following contributions in this work:

- **Inductive Deductive Synthesis (IDS):** the first general technique for jointly co-evolving code and machine-checked proof under a partial-proof oracle, with feedback from failures and performance benchmarks in the same loop.
- **IDS-based multi-agent system:** the first agentic system to autonomously generate *verified* distributed systems. Solves $7/7$ consistency specifications where SOTA agents (GPT-5.4, Opus 4.6) solve only $2/7$, with up to $3\times$ the throughput of published verified systems.

[1] Past work [31] reports 9–12 months of expert effort to write a proof for a key-value (KV) store.

- **IDS suite:** six new Rocq specifications for distributed key-value-store consistency models, released as an open benchmark for verified-synthesis research.

2 Related Work

Verified code generation with an LLM has three components: (1) a specification of desired guarantees, (2) an implementation, and (3) a machine-checkable proof that the implementation satisfies the specification. Prior work falls into four threads: three each address one component (specification generation, program generation with an evaluator, automated proof generation), and verified synthesis attempts (2) and (3) jointly. IDS pursues this last class on distributed-systems specifications.

Specification generation. A complementary line uses LLMs to extract formal specifications from code or intent [16, 64]; Clover [53] generates formal annotations from natural language specifications, and then checks the consistency of code with the formal and natural specifications on Dafny benchmarks. KBSpec [62] generates a Java Modeling Language specification from a knowledge base of documentation and verifier feedback, and SLD-Spec [12] generates an ACSL specification for C [8] via program slicing and logical deletion. IDS takes such specifications as input rather than producing them. Specification generation from reference code is an avenue of further investigation (§6 and App. H).

Program generation with an evaluator. A separate strand of work treats program construction as search guided by evaluator feedback. Coding agents get feedback from given or self-generated tests [70, 50], or from communication with other specialized agents [19, 45, 66]. Algorithmic discovery systems use a scoring function to drive evolution [47, 40] or deep reinforcement learning [36]. IDS shares the evaluator-guided search scheme; however, instead of tests or scores, it uses a proof assistant. In contrast to sampled signals (tests over traces, specific inputs, or estimated scores), type-checked proofs guarantee correctness on all inputs.

Automated proof generation. Most prior LLM-for-verification work fixes the implementation and specification, asking the model only for the proof. For SMT-backed languages such as Dafny [29] and Verus [26], recent systems synthesize loop invariants and assertions for the verifier. They do this either via compiler-feedback loops [68] or targeted assertion generation [39]. For tactic-driven assistants (Lean [14], Rocq [9], Isabelle), provers trained on prior formal proofs [71], natural-language proofs [61], and conjectured theorems [15] either generate full proofs, in one pass [46, 59] or with repair [17, 42, 60], decompose into sub-goals [32, 75, 21], or predict and repair one tactic at a time from proof state and verifier feedback [20, 72, 55]. AutoRocq [58] and ProofCoop [23] likewise take the implementation as given, the latter with an ensemble of custom-trained models. IDS instead starts from no implementation, so proof obligations shift as code evolves; it co-evolves code and proof with off-the-shelf agents, and further optimizes performance.

Verified synthesis. Producing a program and its proof from a specification alone dates to deductive [38, 44] and counterexample-guided inductive synthesis [51, 52] and their combination [48], but these methods' inference rules and search spaces confined results to small functions. LLMs rekindle this dream. VERINA [73] benchmarks spec, code, and proof generation and finds automatic provers wanting; AlphaVerus [1] bootstraps verified Verus code at function level. At the other extreme, hand-built verified systems (IronFleet's Paxos [18], Verdi's Raft [65], FSCQ's crash-safe file system [11], Anvil's Kubernetes controller [54], Chapar's KV store [31], and Iris-based work [22, 49, 10]) show that machine-checked guarantees are achievable at systems scale, but at months to years of expert effort. IDS shows that verified synthesis need not be small-scale or slow: it automatically generates implementations and their machine-checkable correctness proofs for distributed key-value stores in hours rather than months.

3 Background and Overview

We begin with a short introduction to formal verification. A formally verified system has three pieces. The *specification* is a precise mathematical statement of correctness; the *implementation* is the code that runs; and the *proof* is a machine-checkable argument that, on every input, the implementation matches the specification. Historically, humans wrote all three; IDS takes a human-written spec and automatically produces both implementation and proof, using LLM agents driven by the proof assistant Rocq [9]. We show a simple example: a *counter*, a state machine with two operations (`inc` to increment the count and `read` to return it) starting from an initial state `init`.

Specification

```
Module Type CounterSpec.

  Parameter t : Type.
  Parameter init : t.
  Parameter inc : t -> t.
  Parameter read : t -> nat.

  Axiom read_init :
    read init = 0.

  Axiom read_inc :
    forall s,
    read (inc s) =
      S (read s).

End CounterSpec.
```

Partial impl. & proof

```
Definition t := list unit.
Definition init : t := nil.

Definition inc (s : t) : t.
Admitted.

Definition read (s : t) :=
  length s.

Theorem read_init :
  read init = 0.
Proof. reflexivity. Qed.
Theorem read_inc :
  forall s,
  read (inc s) = S (read s).

Admitted.
```

Complete impl. & proof

```
Definition t := list unit.
Definition init : t := nil.

Definition inc (s : t) := tt::s.

Definition read (s : t) :=
  length s.
Theorem read_init :
  read init = 0.
Proof. reflexivity. Qed.
Theorem read_inc :
  forall s,
  read (inc s) = S (read s).

Proof. intros s. unfold read,
     inc.
  simpl. reflexivity. Qed.
```

Figure 2: Counter *specification* (left), *partial* synthesis (center, yellow = `Admitted`), and *complete* synthesis (right, green = filled in). Rocq accepts both files.

Fig. 2 shows the counter's *specification* (left), a *partial synthesis* (center), and a *complete synthesis* (right). The spec states two properties, or "axioms," about `init`, `inc`, and `read`: reading the initial state returns zero, and reading after an increment returns one more than reading before. In the partial synthesis, `inc`'s body and the `read_inc` theorem are deferred via `Admitted` (a placeholder for unfinished work), but Rocq still accepts the file: the chosen representation (a list whose length encodes the count) is consistent with the work so far. The complete synthesis fills in the deferred pieces. The representation is a free choice; any state satisfying the axioms works, though `nat` (natural numbers) is more efficient.

Rocq's type-checker as an oracle, including for partial work. Rocq's checker *accepts* a declaration/proof if and only if it satisfies its stated type/proposition, and rejects with a precise diagnostic otherwise. There are no false positives or false negatives, in contrast to tests (only the inputs tried), static analysis (over-approximate), or LLM-as-judge (guesses). The verdict also extends to *partial* code, as the center column of Fig. 2 illustrates: unproven obligations are stated as *deferred holes* (`Admitted` lemmas or stub function bodies), and the file still type-checks. (App. G walks a longer `all_less_than` example through this progression.) IDS leverages the type-checker as an oracle to drive a *deductive synthesis* [38]: at every step, the agent extends the partial implementation and proof, then asks Rocq whether the partial state still type-checks, thus co-evolving implementation and proof. A yes keeps the design on track; a no rules it out before further work is committed. If progress stalls (measured by the number of partial proofs left empty), IDS reverts to an earlier state.

From counter to distributed stores. IDS applies this loop at scale. For distributed systems, the specification grows from simple axioms to conditions over executions involving messages, clients, replicas, and failures. For example, `read_inc` in a distributed setting becomes read-your-writes: a client that increments then reads must observe its own increments. The implementation grows from a few-line module to a multi-replica protocol with larger state, multiple message types, and handlers. For example, a replica can no longer store one `nat`; it must keep a vector with one entry per client, tracking the increments received from that client. On a request carrying the client's increment count, the replica can then tell whether it is up-to-date enough to serve, providing read-your-writes. The proof grows from simple tactics to an inductive simulation argument (an induction linking implementation and spec states) with dozens of helper lemmas. Such implementations span a large, subtle design space with widely varying performance, explored by decades of research. §4 describes the agent that drives this process, App. I gives the full specifications, and §5 reports results.

4 IDS Agentic Architecture

Overview. We now describe how we realize IDS as an agentic architecture (Fig. 3). Given a system interface and property specifications, the framework orchestrates multiple LLM agents guided by verification and performance feedback to generate an efficient, mechanically-checked implementation. This architecture relies on a powerful synergy: deductive synthesis constructs the code and proof under a given strategy, while inductive synthesis discovers better strategies from failed attempts.

Inspired by deductive synthesis [38, 37], IDS introduces Deductive Synthesis Agents (DSAs, 1) to incrementally synthesize implementations from specifications. Guided by a strategy, a DSA

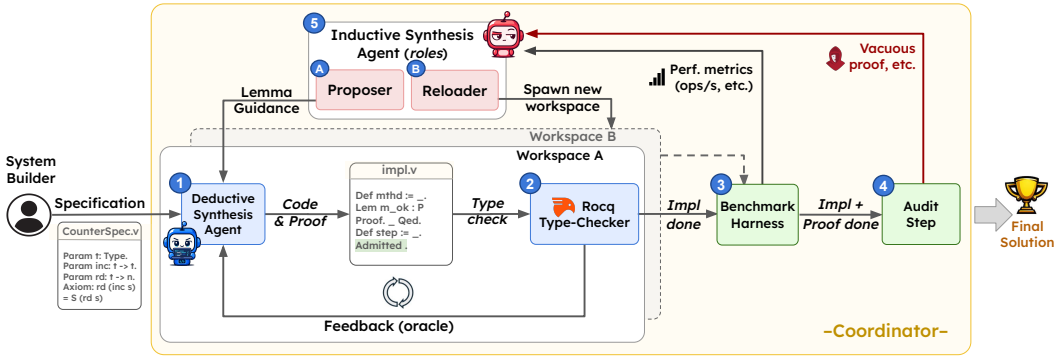


Figure 3: IDS' agentic architecture; code and proof advance jointly throughout. A coordinator runs parallel (1) Deductive Synthesis Agents (DSAs); each step is graded by (2) Rocq's type-checker, completed implementations are (3) benchmarked, and closed proofs are (4) audited for non-vacuity. On failure, the (5) Inductive Synthesis Agent (ISA) intervenes: (A) *proposer* adds helper lemmas on tactical stalls; (B) *reloader* respawns a DSA with a fresh design on strategic dead-ends.

recursively decomposes a component and its specification into sub-components using placeholders. The DSA constructs proofs assuming these sub-components are correct, repeating this process down to trivial elements. The Rocq type-checker (2) continuously verifies both complete and partial proofs throughout this decomposition.

While this deductive approach succeeds when guided by a sound search strategy (e.g., a high-level design or proof approach in the prompt), a DSA can stall under a dead-end strategy. Inspired by inductive synthesis [52, 51, 13], IDS introduces an Inductive Synthesis Agent (ISA, 5) that learns from failed strategies to propose more promising ones. The ISA receives feedback when a DSA stalls or yields poor performance. It then intervenes in two roles: a *proposer* for tactical, local adjustments (A), and a *reloader* for strategic, global shifts to entirely new designs (B).

Coordinator. The coordinator launches and orchestrates the DSAs and the ISA, maintaining overall system state and managing a parallel pool of workers. While DSAs can report localized errors, they cannot detect strategic dead-ends on their own. The coordinator monitors progress to identify stagnating agents, recording failed strategies and invoking the ISA to propose new ones before respawning the DSA.

To ensure the synthesized system is not only correct but also performant, the coordinator benchmarks each candidate eagerly, even before its proof completes. It extracts the implementation to executable code and runs the benchmark harness (3, §5.1). The measurements are fed back to the ISA so that its future strategies guide the DSA toward more efficient implementations. Once a proof closes (i.e., the proof completes with no remaining `Admitted` placeholders), the coordinator audits the result to ensure it is fully verified, non-trivial, and implements the expected interface (4). The audit is a deterministic script, not a model (App. F.3). With it, 20 of 21 runs on the seven key-value store specifications yield a genuine proof and the other does not close; without it (−A in Table 2), 7 are genuine, 9 leave admits or axioms, and 5 are vacuous. Finally, the coordinator returns the most efficient verified solution found within the time budget.

DSA: Deductive Synthesis Agent. Off-the-shelf LLMs, having been pretrained on a large corpus of programs and proofs, are well-equipped to guide synthesis search, as we empirically demonstrate in §5.2. To harness this capability, we design the DSA as a generic LLM coding agent operating under strict constraints: implementations must be well-typed before proving, specifications cannot be altered, `Axiom`, `Hypothesis`, and `Parameter` are forbidden at every step (App. F.2), final outputs must contain no unproven assumptions, and system state must remain bounded.

At each node of the search tree, the agent can take the following actions to advance the synthesis: (1) define partial implementations, (2) close open proof branches, or (3) decompose complex lemmas into simpler helper lemmas. After each step, the implementation and proof are passed to Rocq's type-checker. If there are no errors, the state is saved, building a search tree of well-typed partial

implementations and proofs. If errors occur, the agent can take the following actions: (1) repair the failed step by attempting a different approach, or (2) revert to an earlier node in the search tree.

ISA: Inductive Synthesis Agent. The DSA cannot reliably step back and pivot a failing strategy on its own. The ISA fills this gap as a stateless LLM agent that reasons over a strategy summary and proposes new approaches across two decision horizons:

- **Proposer (tactical).** Triggered when the strategy is promising but the DSA is stuck on a specific proof. The proposer offers finer-grained assistance, suggesting helper lemmas and structural decompositions from a holistic view of the proof state.
- **Reloader (strategic).** Triggered when the strategy is a dead-end or slow. Rather than allowing the DSA to endlessly apply small fixes, the reloader intervenes by respawning the DSA with an entirely new high-level design. Before a reload, the DSA reuses its saved states (repair first, then revert); the reloader writes the new design from the record of past attempts (App. F.2 and F.5).

Within a run, IDS keeps the full history of attempts in persistent files (a design log of every approach tried and why it failed, plus the current status and strategy guidance), which the agents selectively inspect on each call rather than carrying in a single ballooning context session.

5 Evaluation

We evaluate IDS on 7 Rocq specifications of distributed key-value-store consistency: Chapar's published causal-consistency spec [31] and 6 new *IDS suite* specs co-developed with a formal-verification and distributed-systems expert (App. I). We extract synthesized implementations from Gallina to OCaml and benchmark them on a 5-VM Google Cloud cluster under a distributed runtime against hand-written expert references. For synthesis, we compare against SOTA coding agents, Codex (GPT-5.4) [41] and Claude Code (Opus 4.6) [4], under the same prompt and budget. We answer three questions in the following subsections:

- **Q1. Does IDS handle hard specifications?** IDS succeeds on all 7 in about 6.9 hours and \$106 per spec on average (max 11h, \$155), versus 2 of 7 each for Codex and Claude Code under the same prompt and budget (§5.2).
- **Q2. Are the synthesized implementations performant?** They match or beat every expert-implemented reference: up to $3\times$ throughput on Chapar's vector-clock reference, $1.4\times$ on IDS suite CC and monotonic reads, comparable on read-your-writes (within 21%), and sustained throughput on MW and RYW+MW where the reference times out (§5.3).
- **Q3. Which of IDS' design decisions drive these results?** The joint code-and-proof co-design, the ISA's proposer and reloader roles, the audit step, the Rocq feedback to the DSA, and the performance feedback to the ISA each contribute consequentially. The proof-engineering guidance in the DSA's prompt is an efficiency aid: removing it has minimal effect on efficacy but slows the search. The DSA alone also sets a new state-of-the-art on seven public proof benchmarks across four languages (§5.4).

5.1 Experimental Setup

Specifications. Each specification is a high-level reference implementation of a key-value store; its state type and operations define the consistency notion. IDS takes a specification and produces a refining implementation paired with a machine-checked proof of correctness on every input.

Chapar [31] provides a published causal-consistency specification. The other six form the *IDS suite* benchmark dataset that we release alongside the system (App. I): read-your-writes (RYW), monotonic writes (MW), monotonic reads (MR), their composition (RYW+MW), and two causal-consistency variants: IDS suite CC and LCC (*labeled* causal consistency, where each client subscribes to a set of topic labels and sees causality enforced only on those labels). The IDS suite spans the space of session consistency guarantees. To the best of our knowledge, no prior benchmark provides verified Rocq artifacts across this range. Difficulty grows in three tiers: simple (RYW, MW), specs whose proofs need many cases (MR, RYW+MW), and specs that also manage explicit dependency sets (IDS suite CC, LCC). A side-by-side summary of all seven specifications appears in App. A.

Harness and metrics. Verified Rocq implementations are extracted to OCaml (via Rocq's built-in extraction, an automatic compilation pass from Gallina to OCaml) and run under a distributed runtime

Table 1: IDS succeeds on **all seven specifications** ($\geq 2/3$ runs) versus only **2 of 7 each** for Codex and Claude Code, while using **less total wall-clock and fewer total dollars**; on every spec where any method succeeds, IDS is also **faster and cheaper**. **pass rate** = # successes out of 3 (green: $\geq 2/3$; red: $< 2/3$); **hours** = median wall-clock until a successful run (Rocq `Qed` accepted, audit step passed, extracted OCaml runs on the harness); **cost** = median dollars per run. All methods share the same model, verifier, pass criterion, and per-spec budget; relative std across runs is $\leq 7\%$.

	Codex			Claude Code			**IDS**		
Specification	Pass rate↑	Hours↓	Cost ($)↓	Pass rate↑	Hours↓	Cost ($)↓	Pass rate↑	Hours↓	Cost ($)↓
Chapar									
Causal Consistency	0/3	12	160	0/3	11	165	**3/3**	**10**	**148**
IDS suite									
Read-Your-Writes	**3/3**	4	60	**3/3**	3	70	**3/3**	**2**	**52**
Monotonic Reads	0/3	12	130	0/3	10	140	**3/3**	**7**	**102**
Monotonic Writes	2/3	5	65	**3/3**	4	70	**3/3**	**3**	**58**
RYW + MW	0/3	9	125	1/3	6	110	**3/3**	**5**	**93**
Causal Consistency	0/3	12	160	0/3	11	170	**2/3**	**11**	**155**
LCC	0/3	13	180	0/3	11	160	**3/3**	**10**	**136**
Total across specs	2/7	67	$880	2/7	56	$885	**7/7**	**48**	**$744**

on a 5-VM Google Cloud cluster. As in prior verified system works (Verdi [65] and Chapar [31]), for verification guarantees to hold, we assume that extraction, the OCaml toolchain and the runtime are trusted (App. H). Each run executes 4 parallel workers $\times$ 1,000 random operations; we sweep across put rates, and run a separate scaling sweep at put rate $= 50\%$ over $N \in \{1{,}000, 2{,}000, 5{,}000, 20{,}000\}$. App. D covers the cluster topology and toolchain. Every method runs under the same per-spec wall-clock and dollar budget. Runs that exhaust either budget are considered failed runs. Each cell aggregates three independent runs, and we report pass rate (successes out of three), time-to-finish, dollar cost, throughput, p99 latency, peak memory, and ops-per-worker scaling, all as medians. Reported time-to-finish includes every per-step Rocq check; each step recompiles only the edited file, taking about 4 seconds on Chapar, versus 12–17 seconds for a full build (App. F.2).

Baselines. The synthesis baselines are Codex (GPT-5.4) [41] and Claude Code (Opus 4.6) [4]. Both run under the same prompt and budget as IDS' DSA, so the comparison isolates the agentic architecture. Codex is also IDS' primary backend. For performance comparisons, the references are the hand-written expert implementations: Chapar's two published references (vector-clock and list-based) [31], and the expert-written reference released alongside each IDS suite specification, except LCC, a custom spec with no expert reference. For the cross-language evaluation, we adopt seven public verification benchmarks (DafnyBench, Verus-Bench, miniCodeProps, CoqStoq, VerKer, VeriSoftBench, VeruSAGE-Bench) across the languages Dafny, Verus, Lean, and Rocq. We compare against the strongest published prior method for each benchmark. App. E covers three more (VERINA, AlgoVeri, CloverBench). These cross-language benchmarks target function- and algorithm-level tasks, smaller and simpler in scope than the real-world distributed systems above, though VerKer, VeriSoftBench, and VeruSAGE-Bench draw their tasks from the Linux kernel, 23 formal-methods repositories, and eight real-world Verus projects.

5.2 Synthesis Correctness

IDS returns verified implementations for all seven specifications, a $3.5\times$ pass rate over the 2 out of 7 that each of Codex and Claude Code reach under the same prompt and budget. Even on the two specs (RYW and MW) where both baselines succeed, IDS is $1.6\times$ faster and 17% cheaper (Table 1). IDS succeeds by storing the data in a way that lets the proof split into smaller, manageable cases (e.g., one entry per key on Chapar CC, or one entry per (key, client) pair on monotonic reads); the baselines keep the spec's default layout, never backtrack, and stall on the same hard cases. We also evaluate two LLM-only baselines that ask Codex for an implementation only (the standard coding-agent task), with best-of-N=100 generation, on 4 properties (RYW, MW, MR, CC). *Setting 1 (spec-given):* Codex receives the formal Rocq specification. *Setting 2 (vibe coding):* Codex receives only a natural-language paragraph of the property. We then check each selected implementation two ways: an adversarial multi-client trace flags implementations that violate the spec on a single execution, and a refinement proof against the spec catches the rest. Spec-given Codex passes both checks on $1/4$ properties;

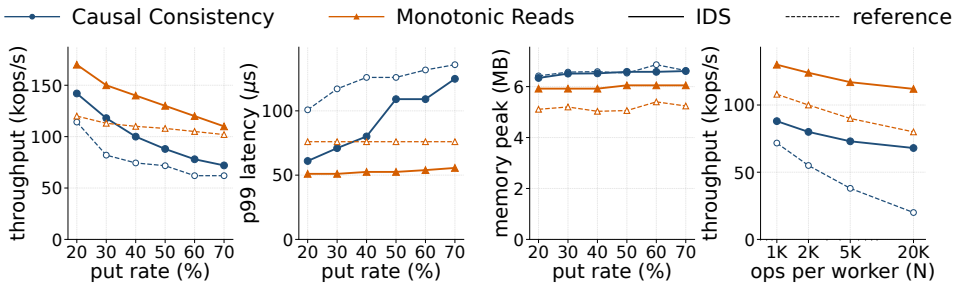


Figure 4: Throughput, p99 latency, and peak memory vs. put rate, and throughput vs. ops-per-worker N, for IDS suite CC and monotonic reads. Solid = IDS; dashed = reference (function-as-map for both). The per-spec breakdown for all seven specifications appears in App. D.1.

vibe coding on $0/4$ (App. C). Even given the formal spec and 100 candidates per property, the LLM alone cannot produce implementations that are guaranteed correct on every input. Conversely, given the spec and IDS' synthesized implementation, Claude Code closes the proof on only 4 of 7 specs, failing Chapar CC, IDS suite CC, and LCC (Table 5). Implementation and proof are thus both hard, even though proofs make up 87–97% of each final artifact's lines (Table 4). Only RYW passes in Setting 1, and only RYW passes a majority of runs without joint discovery (−J, §5.4). The simplest spec closes both times; everything harder needs implementation and proof to advance together.

Chapar causal consistency. IDS closes Chapar's published causal-consistency proof, originally a major hand-written verification [31], in 10 hours and $148 (roughly 200× faster than the cited 9–12 months of expert effort), while both agent baselines fail despite spending more time and money. IDS succeeded by changing how the data was stored. Its first attempt kept each replica's state as one big object, so proving correctness required reasoning about every key and message together, and the proof got stuck. IDS backtracked, gave each key its own small table, and the proof then split into one easy case per key. The full trajectory and the per-family lemma counts are given in App. B.

IDS suite specifications. IDS handles the four harder IDS suite specifications (monotonic reads, RYW+MW, LCC, and IDS suite CC) in 5 to 11 hours per spec, with 3-of-3 pass rates on the first three and 2-of-3 on IDS suite CC. In comparison, Codex never succeeds, and Claude Code succeeds only once, on RYW+MW. The same Chapar pattern recurs: the first design gets stuck on the spec's default layout, then IDS backtracks and rebuilds the data so the proof splits into smaller cases, e.g., one case per message on the two CC variants (App. B).

Failures and cost. IDS recovers from tactical stalls, strategic dead-ends, and vacuous solutions; its only failed run (1 of 21, on IDS suite CC) ended at the budget. Because Rocq checks every proof and the audit filters trivial ones, a failure yields no artifact rather than a false positive (Table 6). Cost scales with the complexity of the guarantee, not artifact size: the session guarantees (RYW, MW, RYW+MW, MR) take 2–7 hours and $52–102, the causal ones (LCC, Chapar CC, IDS suite CC) 10–11 hours and $136–155. LCC is 29% larger than Chapar CC (3,924 vs. 3,037 lines) yet closes in the same 10 hours for less ($136 vs. $148; Tables 1 and 4).

5.3 Runtime Performance

IDS' synthesized implementations match or beat every prior reference: up to 3× throughput on Chapar's vector-clock reference, 1.4× on IDS suite CC and monotonic reads, comparable on read-your-writes (within 21%), and 152k and 139k ops/s on MW and RYW+MW where the reference times out at every put rate (App. D.1). Fig. 4 shows IDS suite CC and monotonic reads vs. the references (full per-spec breakdown in App. D.1). The gap follows from one mechanism: IDS' data-store representations are bounded, while the references' grow with workload size. These representations emerged from the joint code+proof loop: the proof obligation drove the agent to them (§5.2). The performance feedback from the benchmark harness then steers the agent toward the fastest among multiple verifying candidates (§5.4).

IDS suite CC. Both implementations enforce causality with one timestamp per client; the data store differs. The reference's function-from-key-to-value extracts to OCaml as a chain that grows per put (every get walks it), while IDS uses a balanced-tree map where a lookup costs at most the tree depth. IDS reaches up to 1.4× peak throughput at low put rate where gets dominate, with p99

latency up to $1.7\times$ lower. The gap narrows at high put rate as broadcast becomes the bottleneck; tail latency rises with put rate on both implementations as broadcast queue waits grow. Peak memory is comparable at 1k ops per worker, as vector-clock state is bounded by client count rather than put rate; larger workloads would push it upward. On ops-per-worker scaling, the reference's throughput collapses by more than half from 1k to 20k ops per worker while IDS loses only about a quarter.

Other IDS suite specifications. Unlike CC, these specs only need per-client guarantees, so they do not pay CC's per-write cross-client coordination cost; their tail latency stays flat across put rate (Fig. 4, MR). On the other IDS suite specs, IDS ranges from comparable performance on read-your-writes (chain stays short) to a large gap with the reference timing out at every put rate on the monotonic-write specs (MW, RYW+MW). The four with prior references (RYW, MW, MR, RYW+MW) share the closure-chain mechanism: the reference's function-from-key-to-value extracts as a chain that grows per put; IDS replaces it with a flat list or balanced tree, and the speedup tracks how often the chain is walked. On the monotonic reads benchmark, a get in the reference walks every operation, so IDS outperforms it by $1.4\times$.

5.4 Ablations and Generalization

The joint step-wise discovery of code and proof is the key design decision behind IDS. We ablate this joint design itself ($-$J), the ISA's proposer ($-$P) and reloader ($-$R), the coordinator's audit step ($-$A), and the Rocq feedback to the DSA ($-$VF), as well as the proof-engineering guidance in the DSA's prompt ($-$G) across the seven key-value store specs and VERINA's 189 Lean tasks (Table 2).

Architecture ablations. Without joint discovery, only RYW succeeds in a majority of runs: Chapar CC, RYW+MW, IDS suite CC, and LCC drop to 0 out of 3, and VERINA falls to 85% (-8% from full IDS' 93%). This is because a fixed implementation limits the proof phase. On Chapar CC, $-$J commits to the global per-replica state closest to the spec. Full IDS reloads to per-key entries when the proof stalls. The audit step catches trivial solutions that pass Rocq's type-checker: on IDS suite CC, an agent without auditing once shipped a `put-guard` returning `false` unconditionally, with the theorem being trivially satisfied. On the other hand, cells under $-$A count only audited successful runs. Ablations of the proposer and reloader each drop pass rate on the four hardest specs (Chapar CC, MR, IDS suite CC, LCC), taking VERINA to 79% (-14%) under $-$P and to 88% (-5%) under $-$R. The Rocq feedback to the DSA is the most consequential single component: replacing the structured Rocq diagnostic (goal, hypotheses, tactic backtrace) with only an accept/reject drops every spec to at most 1 of 3 successful runs and VERINA to 58% (-35%). Removing the proof-engineering guidance from the DSA's prompt ($-$G; its tactical playbook, App. F.2) has minimal effect on efficacy (19 vs. 20 of 21 runs; VERINA 90%, -3%): the agent uses Rocq feedback to rediscover the needed tactics itself, only slower, raising mean time from 6.9 to 8.4 hours and mean cost from \$106 to \$144. This guidance is thus an efficiency aid.

Performance-feedback ablation. Without performance feedback ($-$PF), the agent commits to the first implementation and proof that passes the type-checker and misses the course-corrections seen in §5.3. As Fig. 5 shows, full IDS is on average $1.42\times$ faster than $-$PF across the six specs with reference baselines (Chapar CC, RYW, MW, RYW+MW, MR, IDS suite CC).

DSA as a general prover. The DSA's effectiveness is not specific to IDS' synthesis loop or to Rocq: applied alone to public verification benchmarks across four languages, it sets a new state-of-the-art on each (Fig. 6, Table 9). It saturates miniCodeProps (100 of 100) and Verus-Bench (149 of 150), and beats prior SOTA by 36 to 69% on DafnyBench and CoqStoq. IDS' agentic architecture brings further gain: full IDS adds $+10\%$ on DafnyBench, $+20\%$ on miniCodeProps, and $+51\%$ on CoqStoq over Codex under the same prompt, while a baseline-model swap (Codex $\rightarrow$ Claude under the same prompt, no IDS architecture) gains at most 6% on any of these four benchmarks. On VerKer, VeriSoftBench, and VeruSAGE-Bench, IDS solves 39 of 60 (AutoRocq [58] 18), 432 of 500 (prior SOTA 205), and 838 of 849 (prior SOTA 797). On VeruSAGE-Bench, IDS leads, confirming that it generalizes to Rust systems such as OS kernels, memory allocators, and serialization code. IDS also reaches state-of-the-art on all three code-and-proof benchmarks: $176/189$ on VERINA (prior SOTA 38), $65/77$ on AlgoVeri ($2\times$ prior SOTA 31), and a full $62/62$ on CloverBench (Table 9; detail in App. E). Beyond per-task benchmarks, IDS synthesizes the lock of IronFleet [18] in Dafny, verified by Z3 rather than Rocq: from the unmodified spec, it regenerates and proves all 15 files with no axioms or assumptions (App. E.3).

	full	−J	−P	−R	−A	−VF	−G
Chapar CC	3/3	0/3	0/3	0/3	1/3	0/3	3/3
RYW	3/3	2/3	2/3	3/3	3/3	1/3	3/3
MW	3/3	1/3	2/3	2/3	2/3	0/3	3/3
RYW+MW	3/3	0/3	1/3	2/3	1/3	0/3	3/3
MR	3/3	1/3	0/3	1/3	0/3	0/3	3/3
IDS suite CC	2/3	0/3	0/3	0/3	0/3	0/3	2/3
LCC	3/3	0/3	0/3	0/3	0/3	0/3	2/3
VERINA	176	160	150	167	176	110	171

Table 2: Component ablations: # successes from 3 runs (189 tasks for VERINA).

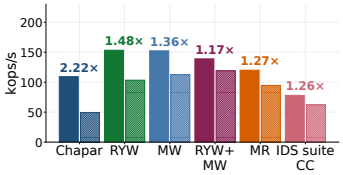


Figure 5: Performance-feedback ablation at *put* = 60%: IDS (solid) vs. no-feedback (hatched).

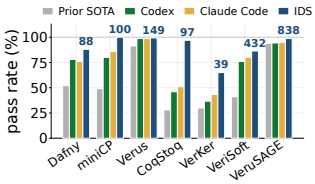


Figure 6: Cross-language success rate (%) on seven verification benchmarks; labels are # of successes.

6 Discussion and Conclusion

IDS is language- and problem-agnostic. IDS needs only (i) a checker that accepts partial proofs, (ii) a programmatic interface an agent can drive, and (iii) extraction to executable code for performance feedback. Any language with a native verifier qualifies (we already run on Dafny, Lean, Verus, and Rocq; §5.4 and App. F.2), as does any language with a reusable formal semantics in a proof assistant (e.g., CompCert's C semantics [30]). A domain needs only a formal specification and, optionally, a performance benchmark. We chose Rocq for its mature ecosystem, and key-value store specifications because the literature already provided one, which we extend with six more (App. I).

Limitations. Our work has three main limitations. First, IDS requires a formal specification as input; writing complete formal specifications for distributed systems requires substantial expert effort. Still, this effort is small relative to the proof effort IDS replaces: our six specifications (App. A) took a few days and are reusable, and the audit rules (App. F.3), which took less effort, are shared infrastructure rather than a per-task cost. Second, the seven specifications we evaluate cover correctness-critical behavior but not the full operational envelope of a production KV store; they do not define interfaces for scaling out (adding or removing nodes at runtime), reconfiguration, fault recovery, or informal but practically necessary behavior such as logging and observability endpoints. Given a richer specification that captures these aspects, we see no reason IDS would not apply unchanged, but evaluating this is left to future work. Third, our empirical evaluation of system-level synthesis covers the distributed key-value store problem and the IronFleet lock [18] only, leaving validation on other domains (e.g., OS protocols, cryptographic primitives) to future work. App. H discusses additional limitations and broader impacts.

The specification bottleneck. While IDS automates implementation and proof synthesis, writing the specification remains the largest open problem: capturing intended behavior in machine-checkable form is hard, and a guarantee is only as strong as the spec itself [25]. We envision two directions for closing this gap: *adversarial specification synthesis*, in which an agent interrogates a human stakeholder in natural language, surfacing ambiguities until a formal spec emerges, and *specification extraction*, where an agent recovers the design-agnostic correctness properties of a deployed system so IDS can synthesize faster implementations that satisfy them. We leave both to future work.

Memory across runs. IDS treats each run as independent, but lessons from one spec can carry over to related ones. To test this, we record what worked on RYW (the winning abstraction, invariants, and lemma decomposition) in a *skill card* and give it to the agent when it starts RYW+MW. The card only guides the search; Rocq still checks every proof. With this guidance, RYW+MW still passes in $3/3$ trials, but considerably faster: in 3 hours for \$62 instead of 5 hours for \$93 (Table 7). This works because key-value store consistency guarantees share structure. Storing such cards in a vector database would let IDS reuse them across many specs; we leave this to future work.

Conclusion. Verified systems construction has traditionally required months to years of expert effort; IDS condenses it to hours by giving an LLM coding agent fast, exact correctness feedback and turning joint code-and-proof construction into a search problem. This shifts vibe coding to verified coding: systems built by LLMs and machine-checked against a formal specification, thus eliminating silent bugs against the specification. Crucially, IDS can apply to any domain with a machine-checkable correctness oracle, including OS kernels, compilers, cryptographic protocols, and hardware. Ultimately, IDS shows that verified software generation is transitioning from a human-labor bottleneck to a compute-driven approach, paving the way for the broad adoption of verified system synthesis.

Acknowledgments

This research is supported by NSF (IFML) CCF-2019844 and gifts from Accenture, AMD, Anyscale, Broadcom Inc., Google, IBM, Intel, Intesa Sanpaolo, Lambda, Mibura Inc., Samsung SDS, and SAP. We thank our Sky Lab and NetSys colleagues at UC Berkeley for fruitful discussions that shaped this paper. The authors declare no competing interests.

References

[1] Pranjal Aggarwal, Bryan Parno, and Sean Welleck. AlphaVerus: Bootstrapping formally verified code generation through self-improving translation and treefinement. *ICML*, 2025. arXiv:2412.06176.

[2] Mustaque Ahamad, Gil Neiger, James E Burns, Prince Kohli, and Phillip W Hutto. Causal memory: Definitions, implementation, and programming. *Distributed Computing*, 9(1), 1995.

[3] Abhishek Anand, Andrew Appel, Greg Morrisett, Zoe Paraskevopoulou, Randy Pollack, Olivier Savary Bélanger, Matthieu Sozeau, and Matthew Weaver. CertiCoq: A verified compiler for Coq. In *The Third International Workshop on Coq for Programming Languages (CoqPL)*, 2017.

[4] Anthropic. Claude Code: Anthropic's agentic coding system. `https://www.anthropic.com/product/claude-code`, 2025.

[5] Anysphere. Cursor: The AI code editor. `https://cursor.com`, 2025.

[6] Peter Bailis and Kyle Kingsbury. The network is reliable. *Commun. ACM*, 57(9):48–55, September 2014.

[7] Debangshu Banerjee, Olivier Bouissou, and Stefan Zetzsche. DafnyPro: LLM-assisted automated verification for Dafny programs. *arXiv preprint arXiv:2601.05385*, 2026.

[8] Patrick Baudin, Pascal Cuoq, Jean-Christophe Filliâtre, Claude Marché, Benjamin Monate, Yannick Moy, and Virgile Prevosto. *ANSI/ISO C Specification Language, Version 1.20*, 2024. Accessed: 2026-08-07.

[9] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer, 2004.

[10] Tej Chajed, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Verifying concurrent, crash-safe systems with Perennial. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pages 243–258, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Haogang Chen, Daniel Ziegler, Tej Chajed, Adam Chlipala, M. Frans Kaashoek, and Nickolai Zeldovich. Using Crash Hoare logic for certifying the FSCQ file system. In Ethan L. Miller and Steven Hand, editors, *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*, pages 18–37. ACM, 2015.

[12] Zehan Chen, Long Zhang, Zhiwei Zhang, JingJing Zhang, Ruoyu Zhou, Yulong Shen, JianFeng Ma, and Lin Yang. SLD-Spec: enhancement LLM-assisted specification generation for complex loop functions via program slicing and logical deletion. *arXiv preprint arXiv:2509.09917*, 2025.

[13] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *International Conference on Computer Aided Verification*, pages 154–169. Springer, 2000.

[14] Leonardo de Moura and Sebastian Ullrich. The Lean 4 theorem prover and programming language. In André Platzer and Geoff Sutcliffe, editors, *Automated Deduction — CADE 28 — 28th International Conference on Automated Deduction, Virtual Event, July 12–15, 2021, Proceedings*, volume 12699 of *Lecture Notes in Computer Science*, pages 625–635. Springer, 2021.

[15] Kefan Dong and Tengyu Ma. STP: Self-play LLM theorem provers with iterative conjecturing and proving. In *International Conference on Machine Learning*, pages 14114–14136. PMLR, 2025.

[16] Madeline Endres, Sarah Fakhoury, Saikat Chakraborty, and Shuvendu K. Lahiri. Can large language models transform natural language intent into formal method postconditions? In *Proc. ACM Softw. Eng. (FSE)*, 2024. arXiv:2310.01831.

[17] Emily First, Markus N. Rabe, Talia Ringer, and Yuriy Brun. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2023, pages 1229–1241, New York, NY, USA, 2023. Association for Computing Machinery.

[18] Chris Hawblitzel, Jon Howell, Manos Kapritsos, Jacob R. Lorch, Bryan Parno, Michael Lowell Roberts, Srinath T. V. Setty, and Brian Zill. IronFleet: proving practical distributed systems correct. In *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. ACM, 2015.

[19] Sirui Hong, Mingchen Zhuge, Jiaqi Chen, Xiawu Zheng, Yuheng Cheng, Ceyao Zhang, Jinlin Wang, Zili Wang, Steven Ka Shing Yau, Zijuan Lin, Liyang Zhou, Chenyu Ran, Lingfeng Xiao, Chenglin Wu, and Jürgen Schmidhuber. MetaGPT: Meta programming for a multi-agent collaborative framework. In *ICLR*, 2024. arXiv:2308.00352.

[20] Thomas Hubert, Rishi Mehta, Laurent Sartran, Miklós Z Horváth, Goran Žužić, Eric Wieser, Aja Huang, Julian Schrittwieser, Yannick Schroecker, Hussain Masoom, et al. Olympiad-level formal mathematical reasoning with reinforcement learning. *Nature*, 651(8106):607–613, 2026.

[21] Albert Q. Jiang, Sean Welleck, Jin Peng Zhou, Wenda Li, Jiacheng Liu, Mateja Jamnik, Timothée Lacroix, Yuhuai Wu, and Guillaume Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *ICLR*, 2023. arXiv:2210.12283.

[22] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. In Sriram K. Rajamani and David Walker, editors, *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 637–650. ACM, 2015.

[23] Zhanna Kaufman, Emily First, Alex Sanchez-Stern, Kyle Thompson, Sorin Lerner, and Yuriy Brun. ProofCoop: Collaborative automated formal verification. In *Proceedings of the 2026 IEEE/ACM 48th International Conference on Software Engineering*, pages 1327–1340, 2026.

[24] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David A. Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles, SOSP 2009, Big Sky, Montana, USA, October 11–14, 2009*, pages 207–220. ACM, 2009.

[25] Alexander Krentsel, Shubham Agarwal, Mert Cemri, Shu Liu, Sidharth Sankhe, Ziming Mao, Matei Zaharia, and Ion Stoica. Reality is the final verifier: On two key gaps in agentic software engineering. arXiv preprint arXiv:2609.12039, 2026.

[26] Andrea Lattuada, Travis Hance, Jay Bosamiya, Matthias Brun, Chanhee Cho, Hayley LeBlanc, Pranav Srinivasan, Reto Achermann, Tej Chajed, Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Oded Padon, and Bryan Parno. Verus: A practical foundation for systems verification. In Emmett Witchel, Christopher J. Rossbach, Andrea C. Arpaci-Dusseau, and Kimberly Keeton, editors, *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles, SOSP 2024, Austin, TX, USA, November 4-6, 2024*, pages 438–454. ACM, 2024.

[27] Lean Community. lean-eval: Comparator-based Lean formal mathematics evaluation. https://github.com/leanprover/lean-eval, 2026.

[28] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 517–530, 2016.

[29] K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In Edmund M. Clarke and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning — 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers*, volume 6355 of *Lecture Notes in Computer Science*, pages 348–370. Springer, 2010.

[30] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[31] Mohsen Lesani, Christian J. Bell, and Adam Chlipala. Chapar: certified causally consistent distributed key-value stores. In Rastislav Bodík and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 357–370. ACM, 2016.

[32] Zenan Li, Ziran Yang, Deyuan He, Haoyu Zhao, Andrew Zhao, Shange Tang, Kaiyu Yang, Aarti Gupta, Zhendong Su, and Chi Jin. Goedel-Code-Prover: Hierarchical proof search for open state-of-the-art code verification. In *COLM*, 2026. arXiv:2603.19329.

[33] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. Stronger semantics for low-latency geo-replicated storage. In *Proc. NSDI*, 2013.

[34] Evan Lohn and Sean Welleck. miniCodeProps: A minimal benchmark for proving code properties. 2024. arXiv:2406.11915.

[35] Chloe Loughridge, Qinyi Sun, Seth Ahrenbach, Federico Cassano, Chuyue Sun, Ying Sheng, Anish Mudide, Md Rakib Hossain Misu, Nada Amin, and Max Tegmark. DafnyBench: A benchmark for formal software verification. *Transactions on Machine Learning Research*, 2025. arXiv:2406.08467.

[36] Daniel J. Mankowitz, Andrea Michi, Anton Zhernov, Marco Gelmi, Marco Selvi, Cosmin Paduraru, Edouard Leurent, Shariq Iqbal, Jean-Baptiste Lespiau, Alex Ahern, Thomas Köppe, Kevin Millikin, Stephen Gaffney, Sophie Elster, Jackson Broshear, Chris Gamble, Kieran Milan, Robert Tung, Minjae Hwang, A. Taylan Cemgil, Mohammadamin Barekatain, Yujia Li, Amol Mandhane, Thomas Hubert, Julian Schrittwieser, Demis Hassabis, Pushmeet Kohli, Martin A. Riedmiller, Oriol Vinyals, and David Silver. Faster sorting algorithms discovered using deep reinforcement learning. *Nature*, 618(7964):257–263, 2023.

[37] Zohar Manna and Richard Waldinger. Synthesis: Dreams → programs. *IEEE Transactions on Software Engineering*, SE-5(4):294–328, 1979.

[38] Zohar Manna and Richard Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2(1):90–121, 1980.

[39] Eric Mugnier, Emmanuel Anaya Gonzalez, Nadia Polikarpova, Ranjit Jhala, and Yuanyuan Zhou. Laurel: Unblocking automated verification with large language models. *Proceedings of the ACM on Programming Languages*, 9(OOPSLA1):1519–1545, 2025.

[40] Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A coding agent for scientific and algorithmic discovery. 2025. arXiv:2506.13131.

[41] OpenAI. Codex: Lightweight coding agent that runs in your terminal. https://github.com/openai/codex, 2025.

[42] Azim Ospanov, Farzan Farnia, and Roozbeh Yousefzadeh. APOLLO: Automated LLM and Lean collaboration for advanced formal reasoning. In *NeurIPS*, 2025. arXiv:2505.05758.

[43] Thanumalayan Sankaranarayana Pillai, Vijay Chidambaram, Ramnatthan Alagappan, Samer Al-Kiswany, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. All file systems are not created equal: on the complexity of crafting crash-consistent applications. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 433–448, USA, 2014. USENIX Association.

[44] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 522–538. ACM, 2016.

[45] Chen Qian, Wei Liu, Hongzhang Liu, Nuo Chen, Yufan Dang, Jiahao Li, Cheng Yang, Weize Chen, Yusheng Su, Xin Cong, Juyuan Xu, Dahai Li, Zhiyuan Liu, and Maosong Sun. ChatDev: Communicative agents for software development. In *ACL*, 2024. arXiv:2307.07924.

[46] Z. Z. Ren, Zhihong Shao, Junxiao Song, Huajian Xin, Haocheng Wang, Wanjia Zhao, Liyue Zhang, Zhe Fu, Qihao Zhu, Dejian Yang, Z. F. Wu, Zhibin Gou, Shirong Ma, Hongxuan Tang, Yuxuan Liu, Wenjun Gao, Daya Guo, and Chong Ruan. DeepSeek-Prover-V2: Advancing formal mathematical reasoning via reinforcement learning for subgoal decomposition. 2025. arXiv:2504.21801.

[47] Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[48] Sanjit A. Seshia. Combining induction, deduction, and structure for verification and synthesis. *Proceedings of the IEEE*, 103(11):2036–2051, 2015.

[49] Upamanyu Sharma, Ralf Jung, Joseph Tassarotti, M. Frans Kaashoek, and Nickolai Zeldovich. Grove: a separation-logic library for verifying distributed systems. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace, editors, *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pages 113–129. ACM, 2023.

[50] Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36:8634–8652, 2023.

[51] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, UC Berkeley, 2008.

[52] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit A. Seshia, and Vijay A. Saraswat. Combinatorial sketching for finite programs. In John Paul Shen and Margaret Martonosi, editors, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, pages 404–415. ACM, 2006.

[53] Chuyue Sun, Ying Sheng, Oded Padon, and Clark Barrett. Clover: Closed-loop verifiable code generation. In *SAIV*, 2024. arXiv:2310.17807.

[54] Xudong Sun, Wenjie Ma, Jiawei Tyler Gu, Zicheng Ma, Tej Chajed, Jon Howell, Andrea Lattuada, Oded Padon, Lalith Suresh, Adriana Szekeres, and Tianyin Xu. Anvil: Verifying liveness of cluster management controllers. In Ada Gavrilovska and Douglas B. Terry, editors, *18th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2024, Santa Clara, CA, USA, July 10-12, 2024*, pages 649–666. USENIX Association, 2024.

[55] Amitayush Thakur, George Tsoukalas, Yeming Wen, Jimmy Xin, and Swarat Chaudhuri. An in-context learning agent for formal theorem-proving. In *COLM*, 2024. arXiv:2310.04353.

[56] Kyle Thompson, Nuno Saavedra, Pedro Carrott, Kevin Fisher, Alex Sanchez-Stern, Yuriy Brun, João F. Ferreira, Sorin Lerner, and Emily First. Rango: Adaptive retrieval-augmented proving for automated software verification. In *ICSE*, 2025. arXiv:2412.14063.

[57] George Tsoukalas, Jasper Lee, John Jennings, Jimmy Xin, Michelle Ding, Michael Jennings, Amitayush Thakur, and Swarat Chaudhuri. PutnamBench: Evaluating neural theorem-provers on the Putnam mathematical competition. *Advances in Neural Information Processing Systems*, 37:11545–11569, 2024.

[58] Haoxin Tu, Huan Zhao, Yahui Song, Mehtab Zafar, Ruijie Meng, and Abhik Roychoudhury. Agentic verification of software systems. *Proceedings of the ACM on Software Engineering*, 3(FSE):3558–3581, 2026.

[59] Haiming Wang, Mert Unsal, Xiaohan Lin, Mantas Baksys, Junqi Liu, Marco Dos Santos, Flood Sung, Marina Vinyes, Zhenzhe Ying, Zekai Zhu, Jianqiao Lu, Hugues de Saxcé, Bolton Bailey, Chendong Song, Chenjun Xiao, Dehao Zhang, Ebony Zhang, Frederick Pu, Han Zhu, Jiawei Liu, Jonas Bayer, Julien Michel, Longhui Yu, Léo Dreyfus-Schmidt, Lewis Tunstall, Luigi Pagani, Moreira Machado, Pauline Bourigault, Ran Wang, Stanislas Polu, Thibaut Barroyer, Wen-Ding Li, Yazhe Niu, Yann Fleureau, Yangyang Hu, Zhouliang Yu, Zihan Wang, Zhilin Yang, Zhengying Liu, and Jia Li. Kimina-Prover Preview: Towards large formal reasoning models with reinforcement learning. 2025. arXiv:2504.11354.

[60] Ruida Wang, Rui Pan, Yuxin Li, Jipeng Zhang, Yizhen Jia, Shizhe Diao, Renjie Pi, Junjie Hu, and Tong Zhang. MA-LoT: Model-collaboration Lean-based long chain-of-thought reasoning enhances formal theorem proving. In *Forty-second International Conference on Machine Learning*, 2025.

[61] Ruida Wang, Jipeng Zhang, Yizhen Jia, Rui Pan, Shizhe Diao, Renjie Pi, and Tong Zhang. TheoremLlama: Transforming general-purpose LLMs into Lean4 experts. In *2024 Conference on Empirical Methods in Natural Language Processing, EMNLP 2024*, pages 11953–11974. Association for Computational Linguistics (ACL), 2024.

[62] Wenhan Wang and Zeyu Sun. KBSpec: LLM-driven formal specification generation with evolving domain knowledge base. *arXiv preprint arXiv:2606.21339*, 2026.

[63] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed Chi, Quoc V Le, and Denny Zhou. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems*, volume 35, pages 24824–24837. Curran Associates, Inc., 2022.

[64] Cheng Wen, Jialun Cao, Jie Su, Zhiwu Xu, Shengchao Qin, Mengda He, Haokun Li, Shing-Chi Cheung, and Cong Tian. Enchanting program specification synthesis by large language models using static analysis and program verification. In *Computer Aided Verification (CAV)*, pages 302–328. Springer, 2024. arXiv:2404.00762.

[65] James R. Wilcox, Doug Woos, Pavel Panchekha, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Thomas E. Anderson. Verdi: a framework for implementing and formally verifying distributed systems. In David Grove and Stephen M. Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 357–368. ACM, 2015.

[66] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. AutoGen: Enabling next-gen LLM applications via multi-agent conversations. In *First Conference on Language Modeling*, 2024.

[67] Yutong Xin, Jocelyn Qiaochu Chen, Greg Durrett, and Işil Dillig. VeriSoftBench: Repository-scale formal verification benchmarks for Lean. In *Third Conference on Language Modeling*, 2026.

[68] Chenyuan Yang, Xuheng Li, Md Rakib Hossain Misu, Jianan Yao, Weidong Cui, Yeyun Gong, Chris Hawblitzel, Shuvendu Lahiri, Jacob R. Lorch, Shuai Lu, Fan Yang, Ziqiao Zhou, and Shan Lu. AutoVerus: Automated proof generation for Rust code. In *OOPSLA*, 2025. arXiv:2409.13082.

[69] Chenyuan Yang, Natalie Neamtu, Chris Hawblitzel, Jacob R Lorch, and Shan Lu. VeruSAGE: A study of agent-based verification for Rust systems. *arXiv preprint arXiv:2512.18436*, 2025.

[70] John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. In *NeurIPS*, 2024. arXiv:2405.15793.

[71] Kaiyu Yang and Jia Deng. Learning to prove theorems via interacting with proof assistants. In *ICML*, 2019. arXiv:1905.09381.

[72] Kaiyu Yang, Aidan M. Swope, Alex Gu, Rahul Chalamala, Peiyang Song, Shixing Yu, Saad Godil, Ryan Prenger, and Anima Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *NeurIPS*, 2023. arXiv:2306.15626.

[73] Zhe Ye, Zhengxu Yan, Jingxuan He, Timothe Kasriel, Kaiyu Yang, and Dawn Song. VERINA: Benchmarking verifiable code generation. In *International Conference on Learning Representations (ICLR)*, 2026. arXiv:2505.23135.

[74] Haoyu Zhao, Ziran Yang, Jiawei Li, Deyuan He, Zenan Li, Chi Jin, Venugopal V. Veeravalli, Aarti Gupta, and Sanjeev Arora. AlgoVeri: An aligned benchmark for verified code generation on classical algorithms. In *ICML*, 2026. arXiv:2602.09464.

[75] Xueliang Zhao, Lin Zheng, Haige Bo, Changran Hu, Urmish Thakker, and Lingpeng Kong. SubgoalXL: Subgoal-based expert learning for theorem proving. *arXiv preprint arXiv:2408.11172*, 2024.

Appendix

A Specifications and the IDS Suite Dataset

Of the seven specifications evaluated in §5, one is Chapar's published causal-consistency spec [31] and the other six are IDS suite specs we wrote with a formal-verification and distributed-systems expert (1,018 lines of Rocq across five spec files; RYW+MW is the joint composition of RYW and MW with no separate spec; Table 3). Formal definitions are in App. I, which also gives two reference implementations of IDS suite CC: a per-client vector-clock store and a Lloyd–Freedman-style protocol [33] as a systems-scale reference.

Table 3: The seven specifications. Chapar CC is a published spec; the six IDS suite specs are expert-written for this paper. *Property* summarizes the consistency guarantee; *LoC* is lines of Rocq for the spec definition.

Specification	Family	Property	LoC
Chapar CC	Chapar	cross-client causal consistency, defined over execution traces	n/a
Read-Your-Writes	IDS suite	a client reads its own most recent write	159
Monotonic Writes	IDS suite	a client's writes are observed in order	158
Monotonic Reads	IDS suite	a client's reads respect the order of observed writes	185
RYW+MW	IDS suite	composition of RYW and MW (no separate spec)	n/a
CC	IDS suite	cross-client causal consistency via explicit dependency sets	248
LCC	IDS suite	causal consistency parameterized by per-message labels and per-client interest masks	268

Common shape. Every spec is a Rocq `Module Type` with the same seven method signatures (`getReq`, `getGuard`, `get`, `getRes`, `putReq`, `putGuard`, `put`), a `State` type, an `Update` type, and an operational semantics on client-replica configurations. IDS consumes this shape directly.

Generality. IDS is not tuned to any individual spec: the same agent, prompts, and tools handle every spec we evaluate. Any specification matching the `Module Type` shape above is a candidate target for the same loop.

B Synthesis Results: Per-Spec Artifacts

All seven artifacts pass Rocq's kernel with no `Admitted`, `Axiom`, `Hypothesis`, or `Parameter`. The three CC specs (Chapar CC: 3,037 lines / 79 lemmas; IDS suite CC: 3,807 / 121; LCC: 3,924 / 128) are 3–6× larger than the four session specs because their proofs span all replicas, not just sender and receiver.

Table 4: Per-spec implementation choice and proof artifact size. Lemmas = `Qed` count; lines via `wc -l`.

Specification	Implementation choice	Lines				Lemmas
		Total	Impl.	Proof	Proof share	
Chapar CC	per-key store entry with sender clock + dep vector	3,037	100	2,937	97%	79
RYW	per-key (val, client, ts) assoc-list	622	76	546	88%	25
MW	per-key (val, client→ts) assoc-list	722	95	627	87%	30
MR	per-(key, client) assoc-list, replace-on-write	1,063	102	961	90%	30
RYW+MW	per-key (val, client, client→ts) assoc-list	958	96	862	90%	28
IDS suite CC	vector clock + per-key VC map + balanced-tree store	3,807	126	3,681	97%	121
LCC	vector clock + per-key VC map + balanced-tree store + per-client label-set filter	3,924	126	3,798	97%	128

Implementation rationale. **Chapar CC.** The 79 lemmas split: 11 per-method refinement (one per `get`/`put`/`update`/`guard`/`init`), 11 reachable-state invariants, 7 causality preservation, 39 clock and per-key store algebra, and 11 execution-model helpers.

Session guarantees (RYW, MW, MR, RYW+MW). All four share the same assoc-list skeleton; each parameterizes the per-entry payload to its consistency property: RYW carries (val, client, ts) to detect own writes, MW extends to (val, client→ts) to enforce writer order, MR uses one entry per (key, client) updated in place, RYW+MW combines both. The proof technique is shared too: each session uses heavy case-analysis via `Nat.eq_dec` on key and client equality (80–100 splits per spec), varying only in which payload field the splits land on. Each Get scans only the distinct keys touched, not the full Put history. MR is the hardest session spec because its Put proof splits into four sub-cases (same vs different key, same vs different client), each closed by case-analysis on `Nat.eq_dec`.

IDS suite CC. A vector clock, a per-key vector-clock map, and a balanced-tree store; the balanced tree keeps Get logarithmic in distinct keys rather than linear in prior Puts. Of the 121 lemmas: 8 are the step-by-step simulation, 32 are refinement-relation invariants for the state components (cell, replica, snapshot) and their preservation under updates, 19 are vector-clock algebra; the remaining 62 are update, lookup, owner, and dependency-tracking algebra. Get's proof closes by applying the relation that ties the implementation's vector clocks plus per-client owner lists (per-key tracking of which client most recently wrote each entry) to the spec's dependency sets.

LCC. A vector clock, a per-key vector-clock map, and a balanced-tree store, with dependency entries tagged with a label (a 4-tuple of client, timestamp, key, label) and projections at Get filtered by the requesting client's interest mask. The proof closes by step-by-step simulation; the simulation relation ties the implementation's label-tagged vector-clock representation to the spec's label-filtered dependency sets. Of the 128 lemmas: 6 are the step-by-step simulation, 25 are refinement-relation invariants and extensions for state components (cells, replicas, snapshots, key-snapshots), 19 are vector-clock algebra; the remaining 78 are update, lookup, owner, and dependency-tracking algebra.

Synthesis trajectories. The artifacts above are not the first design IDS attempted. In each case, the proof stalled on a goal that resisted splitting under the spec's natural state representation; the ISA's reloader abandoned the branch and chose a representation under which the goal decomposed.

Chapar CC. The hardest case in any causal-consistency proof shows that whenever a replica delivers a remote message, it has already observed every update the message transitively depends on. IDS' first attempt kept the global state representation closest to the spec, under which the delivery lemma had to range over every in-flight message at every replica together, and the DSA could not split it. After the search stalled, the reloader abandoned that branch and a fresh DSA rebuilt on a per-key store with per-entry sender, sender-clock, and dependency vector. The proposer then split the delivery invariant into three sub-lemma classes (per-message, per-store-entry, per-replica-clock), each closing under the new representation. The representation was chosen by what split the proof, not by performance; the per-put performance win reported in §5.3 is a consequence.

Monotonic reads. Both agent baselines, and IDS' first attempt, stalled on the function-from-key-to-value representation. After the reloader abandoned that branch, the agent revised the design to a flat list of per-(key, client) entries updated in place, after which the put proof split into four sub-cases on key and client equality.

IDS suite CC and LCC. The winning move is to break the simulation goal into per-message and per-store-entry sub-lemmas, each closing against the per-key store and label-tagged dependency entries respectively. The baselines try the combined goal in one shot and exhaust the budget.

Proof-only baseline. Table 5 details the proof-only baseline from §5.2. Given the implementation, Claude Code reaches a majority of runs on MR ($0/3 \rightarrow 2/3$) and RYW+MW ($1/3 \rightarrow 3/3$). IDS closes all seven because it discovers implementation and proof jointly. A finished implementation records the final state representation but not the trial and error behind it; in IDS, that search closes the proof.

Failure modes. Table 6 details the failure modes from §5.2, how IDS responds to each, and the ablation that isolates it.

Table 5: Proof-only baseline: successes from 3 runs per spec; *Total* counts specs passing at least $2/3$. *Spec only* is the Table 1 setting, where the agent writes both implementation and proof; with *spec + IDS impl.*, Claude Code also receives the implementation IDS synthesized and writes only the proof.

	Spec only		Spec + IDS impl.	Spec only
Specification	Codex	Claude Code	Claude Code	IDS
RYW	3/3	3/3	3/3	3/3
MW	2/3	3/3	3/3	3/3
MR	0/3	0/3	2/3	3/3
RYW+MW	0/3	1/3	3/3	3/3
Chapar CC	0/3	0/3	0/3	3/3
IDS suite CC	0/3	0/3	0/3	2/3
LCC	0/3	0/3	0/3	3/3
Total	2/7	2/7	4/7	7/7

Table 6: Failure modes, how IDS responds, and the ablation (Table 2) that isolates each. IDS recovers from the first three; the fourth ends the run at the budget.

Failure mode	Example	How IDS responds	Ablation (Table 2)
Tactical stall	On Chapar CC the delivery invariant would not close as a single piece.	The proposer splits it into smaller lemmas (per-message, per-store-entry, per-replica-clock), each of which then closes (App. F.4).	Without the proposer: Chapar CC, MR, IDS suite CC, LCC fall to $0/3$.
Strategic dead-end	On Chapar CC the initial global-state design left the delivery lemma unprovable.	The reloader drops that design and the DSA rebuilds on a per-key store, reaching a proof the first design could not (App. F.5).	Without the reloader: Chapar CC, IDS suite CC, LCC fall to $0/3$; MR to $1/3$.
Vacuous solution	On IDS suite CC one run made the put guard always return false, passing Rocq vacuously (§5.4).	The audit rejects these before they count as solved, so the ISA returns to design for a genuine implementation and proof (App. F.3).	Without the audit: MR, IDS suite CC, LCC fall to $0/3$.
Budget exhaustion	The hardest specs hit the per-spec time cap before the proof closes.	The run is reported as a failure at the cap.	Not an ablation: 1 of 21 KV runs and 41 of 778 tasks on seven of the ten benchmarks hit the cap (App. E).

Memory across runs. Table 7 details the skill-card experiment from §6. Each card is stored under its spec type, and a new spec receives the most similar card in its prompt. The card only guides the search: Rocq still checks every proof, so soundness is unaffected.

Table 7: RYW+MW with and without the skill card from RYW (same model and budget). The no-card row is the IDS result in Table 1.

	Pass rate↑	Hours↓	Cost ($)↓
No card	3/3	5	93
Card from RYW	3/3	**3**	**62**

C Implementing Distributed-KV Specifications with Coding Agents (No Proof)

Setup. We test Codex (`gpt-5.4`) on four KV-store consistency properties (RYW, MW, MR, and CC). For each property, Codex generates 100 candidate implementations under two prompt conditions,

then picks its single best candidate. That selected implementation is the agent's output for that (property, condition).

Setting (1) Spec-Given. Codex receives the formal Rocq specification and the module-type signature.

Setting (2) NL-Only ("vibe coding"). Codex receives only a one-paragraph natural-language description of the property (reproduced verbatim below) and the module-type signature.

Each of the 100 generation sessions runs independently and can compile, edit, and test its own code. The selector pass is also Codex, with the same compile and test access. The selected implementation must use bounded state, cannot copy any reference implementation, and cannot use `Admitted`, `Axiom`, `Hypothesis`, or `Parameter` (so the agent cannot stub out unfilled work). The agent does not write any refinement proof.

We evaluate the selected implementation in two stages. *Stage 1 (test).* We run the implementation on an adversarial multi-client scenario; if it accepts an operation the abstract specification rejects on this scenario, Stage 1 fails. *Stage 2 (proof).* If Stage 1 passes, IDS attempts the refinement proof against the abstract specification. If IDS cannot close the proof within budget, Stage 2 fails. We split into two stages because proof attempts are much more expensive than scenario testing. If Stage 1 fails, Stage 2 fails too: an implementation that breaks the spec on a single trace cannot be proven correct on every trace.

Property statements (LLM-visible in Setting 2).

RYW (Read-Your-Writes). Each client must observe its own previous writes. If a client writes a value to a key and later reads that key in the same session, the read must return the client's value or a more recent one.

MW (Monotonic Writes). Writes by the same client are applied at every replica in the order the client issued them.

MR (Monotonic Reads). Successive reads by the same client observe non-decreasing replica states: once a client has seen a value, later reads must not appear older.

CC (Causal Consistency). If operation A causally precedes operation B (either A comes before B in the same client's session, or B reads a value written by A), every client that sees B must also have seen A.

Table 8: Outcome of best-of-N=100 implementation-only synthesis with Codex. **1** = passed, **0** = failed. Stage 1: the selected implementation runs through a fixed adversarial multi-client scenario without violating the abstract specification on that scenario. Stage 2: IDS builds a refinement proof of the implementation against the abstract specification within budget. Cost is the `gpt-5.4` token cost for the 100 generations plus the selection pass.

	Setting (1) Spec-Given			Setting (2) NL-Only		
Spec	Stage 1 pass	Stage 2 pass	Cost	Stage 1 pass	Stage 2 pass	Cost
RYW	1	1	\$30.50	1	0	\$25.77
MW	1	0	\$32.10	1	0	\$27.33
MR	0	0	\$35.20	0	0	\$29.91
CC	0	0	\$47.80	0	0	\$40.79

Result and takeaway. Best-of-$N = 100$ produces a verified implementation on 1 of 4 properties when given the formal specification (Setting 1, RYW), and on 0 of 4 from natural language alone (Setting 2). The two simpler session properties (RYW and MW) consistently pass the scenario test in both settings, but only RYW with the formal spec also passes the refinement proof. The harder properties (MR and CC) fail even the scenario test in both settings: the textbook patterns Codex selects (vector-clock dominance, last-write-wins) do not implement the per-key or causal-order behavior these specs demand. Two failure modes appear: (a) implementations that fail on a single adversarial trace, and (b) implementations that pass the trace but cannot be proven correct on every reachable state. IDS avoids both by integrating the refinement proof into synthesis rather than running it as a post-hoc check; in the main evaluation it closes the proof on all 7 specifications.

D Additional Performance Results and Throughput Breakdown

IDS' gains over the reference implementations all share one mechanism: IDS' state representations bound per-op cost by the spec, while the references' per-op cost grows with workload size. App. D.1 measures this on all five IDS suite specs with cluster-runnable implementations (RYW, MR, MW, RYW+MW, IDS suite CC), shows Chapar CC at the top of the figure for comparison, and adds an illustrative LCC row (labeled causal consistency: each client subscribes to a set of topic labels and only sees causality enforced on those). Across the IDS suite, throughput tracks consistency strength (RYW $>$ MW $>$ RYW+MW $>$ MR $>$ LCC $>$ IDS suite CC at put $= 50\%$) because stricter specs require more checking work per operation. Chapar CC and IDS suite CC are both causal-consistency variants but their reference implementations are designed differently (Chapar's reads one position out of a flat list of per-write dependencies, while IDS suite's compares an entire per-client clock vector and atomically updates the writer's slot on every Put), so the two CC rows show different per-op work at the same correctness level. App. D.2 and D.3 decompose the gap on Chapar CC and Monotonic Reads.

Setup details (extending §5.1). Cluster zone: `us-central1-f`. Each VM is an `e2-standard-2` (2 vCPU, 8 GB RAM) on Debian 12. Toolchain: OCaml 4.13.1 (dune 3.10) and Coq 8.18.0. Throughput is the sum across the NW $= 4$ workers of N/t_w, where t_w is each worker's wall-clock time and N is its ops count, taking the median over three runs per cell; the put-rate sweep uses $N = 1000$ (4,000 cluster ops total per run) and the scaling sweep varies $N \in \{1{,}000, 2{,}000, 5{,}000, 20{,}000\}$. Per-operation latency is bracketed inside the runtime around `put_method`/`get_method` and merged across workers; p99 is the 99th percentile of the pooled per-op latency distribution. Peak memory is the worker maximum of `Maximum resident set size` from `/usr/bin/time -v`, taking the median over three runs. Workload parameters: `key_range` $= 50$, `val_range` $= 100{,}000$, fixed seed 42. The cluster uses a Chapar-style UDP runtime (`runtime_doctex.ml`) adapted to the seven-method doctex `AlgDef` interface; we patched a socket bug for OCaml 4.13 / Debian 12.

D.1 Per-protocol curves: all five IDS suite specs

Result. Read-then-write specs cause the reference to exceed our 180 s wall-clock cap at every put rate measured: in Fig. 7 the reference's MW and RYW+MW rows show red $\times$ at every cell, with no measurable throughput. IDS runs cleanly throughout: 152 kops/s on MW and 139 kops/s on RYW+MW at put $= 60\%$. The mechanism is a closure chain: the reference stores per-replica state as a function from key to value (a function-as-map); after extraction to OCaml, each prior `Put` adds a nested closure that a `Get` must unwrap. The read-then-write check re-evaluates this chain on every `Put`; combined with UDP packet reordering and the protocol's no-retry compare-and-swap, throughput collapses to zero. RYW has no read-then-write check, so its reference is competitive with IDS across the put-rate sweep: within 21% in either direction depending on workload mix, with IDS winning at low put rate (put $= 20\%$) and the reference faster at high put rate (put $= 60$–70%). The reversal is a per-Put cost trade: IDS' assoc-list rewrites its spine on every `Put` ($O(K)$ cell allocations, K = unique keys touched), while the reference's function-as-map prepends a single closure layer per `Put` ($O(1)$ heap cell). At low put rate the assoc-list's $O(K)$ `Get` lookup beats the reference's closure-chain walk; at high put rate the per-`Put` allocation cost dominates. MR shows the same chain-walk cost on `Get` (per-layer slope quantified in App. D.3); IDS suite CC follows the same mechanism with IDS' balanced-tree store keeping Get logarithmic in distinct keys. Memory: IDS' assoc-list pays 0.1–1.0 MB more on RYW (per-entry overhead) and runs in the same 5–6 MB envelope as the reference on the other specs where the reference completes; on MW and RYW+MW the reference produces no memory measurement (it never completes a run). All memory traces are at $N = 1{,}000$ ops/worker; IDS suite CC's vector-clock state is $O(\#\text{clients})$, so its trace is flat across the put-rate sweep at this workload size. At substantially larger N or more clients, the message-bus and per-cell deps would push memory upward. The rightmost column (throughput vs N at put $= 50\%$) confirms the mechanism on every spec: the reference's throughput drops as N grows because each new `Put` extends the chain, while IDS' assoc-list and balanced-tree structures keep throughput approximately flat. Across the put-rate sweep, both implementations' throughput falls and p99 latency rises as put-rate increases because each `Put` is broadcast to every replica, so more writes mean more in-flight messages and longer per-op queue waits.

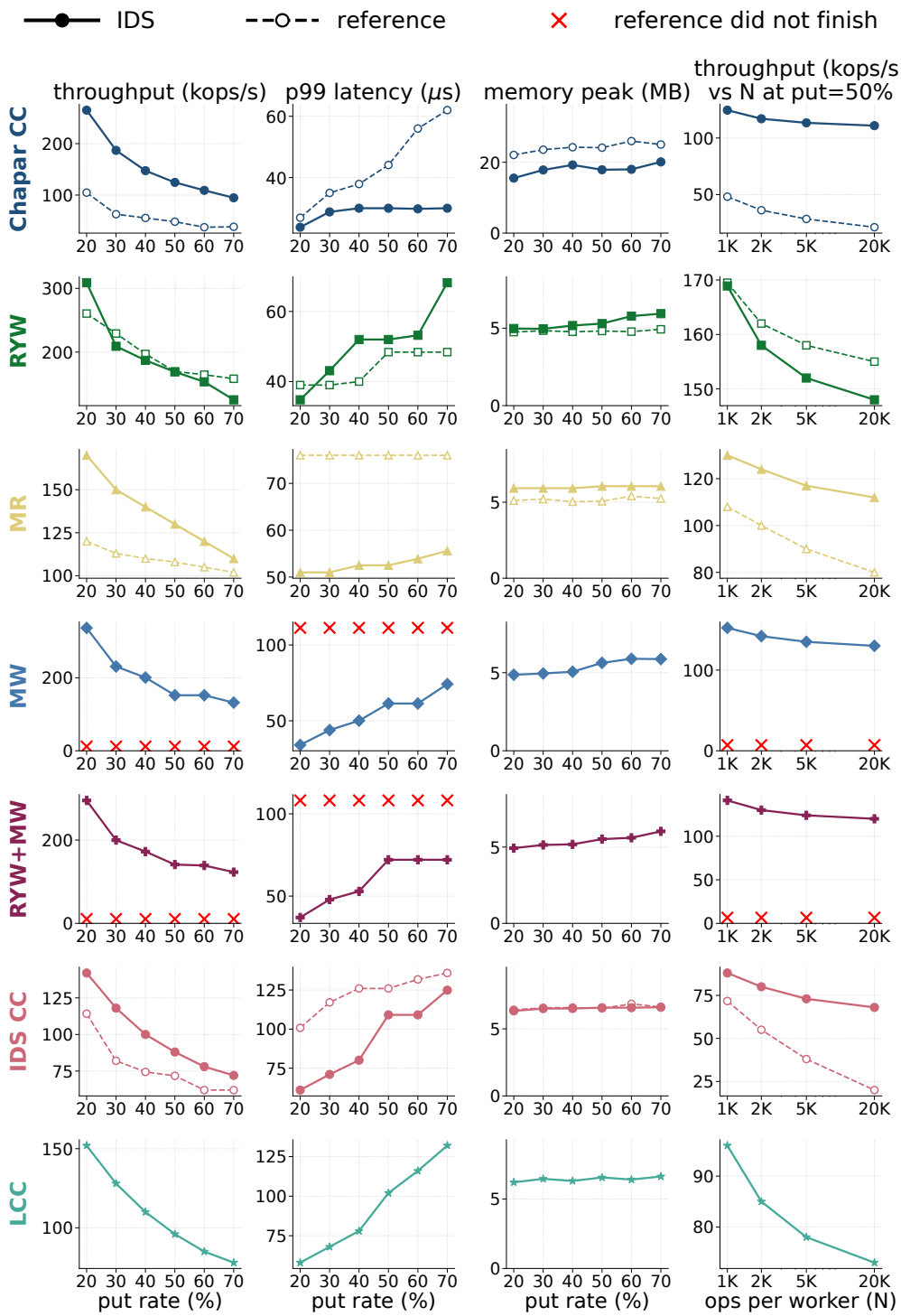


Figure 7: Per-protocol performance. Top row is Chapar CC (Chapar paper protocol) shown for comparison; the next five rows are the IDS suite specs (RYW, MR, MW, RYW+MW, IDS suite CC); the bottom LCC row is a single-line illustrative trajectory of labeled causal consistency. Columns: throughput (kops/s), p99 latency (μs), memory peak (MB), and throughput vs ops-per-worker N at put $= 50\%$. Per-protocol y-axes are dedicated so each row's range is visible. Solid = IDS; dashed = reference (function-as-map). Red $\times$ marks runs that hit the 180 s wall-clock cap; on MW and RYW+MW the reference times out at every put rate (the reference's read-then-write check on a function-as-map is the cause; see App. D.1).

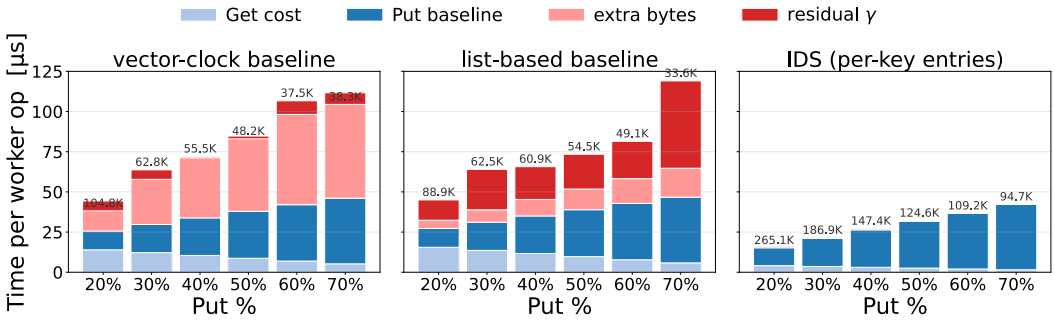


Figure 8: Per-component decomposition of Chapar's two published references (vector-clock, list-based) and IDS' Chapar implementation (per-key store entries, used here as internal pivot). Bar height equals $4/$(published cluster throughput) in μs per worker op. Each bar splits into local Get cost, the per-key-entries Put baseline (same for all), per-algorithm extra wire bytes, and a per-algorithm residual γ. Segments sum to the published value at every cell by construction; where $\gamma < 0$ (e.g., vector-clock at 20, 50, and 70%), the red segment shows $|\gamma|$ and is subtracted from the bar top.

Course correction on MW. Without performance feedback, IDS ships the first verifying design: a balanced-tree store. With performance feedback, the agent runs both candidates in its microbench harness and converges on the assoc-list, which the cluster benchmark harness confirms at 152k ops/s at put $= 60\%$ (Fig. 7). Performance feedback changes the search policy, not the correctness criterion; both candidates are verified IDS suite refinements.

D.2 Throughput breakdown: Chapar CC

Setup. We attribute each algorithm's per-op time to four pieces: its Get cost (varies by algorithm), a shared Put baseline (per-key entries: $58.4\,\mu$s, the smallest of the three), a wire-bytes term proportional to extra bytes per Put ($\beta = 18\,\mu$s per byte, fit from vector-clock vs per-key entries), and a CPU residual γ that captures whatever the wire-bytes term does not explain. Marshaled bytes per Put on the cluster: 44.81 (vector-clock), 41.05 (list-based), 39.62 (per-key entries).

Result. Vector-clock loses on wire bytes; list-based loses on CPU. **Vector-clock** ships $+5.2$ bytes per Put over per-key entries, mapping to $+47\,\mu$s of per-op work at put $= 50\%$; the residual γ is $-1.75\,\mu$s, well below the bytes term. Almost the entire gap is wire bytes. **List-based** ships only $+1.4$ extra bytes per Put on average, yet $\gamma = +21.6\,\mu$s at put $= 50\%$. The list-based replica stores the full history of delivered messages; receivers walk this list on every delivery (causality check), and `Get` walks it again to find the most-recent value for the requested key. Vector-clock and per-key entries do constant-time element-wise lookups.

D.3 Throughput breakdown: Monotonic Reads

Setup. Three MR implementations on the same runtime and identical wire format (92.83 B per `Put` on average): the IDS suite reference (function-as-map), IDS (assoc-list), and a balanced-tree alternative. Throughput differences are CPU-side. The balanced tree is a control: it tests whether IDS' win comes from the assoc-list specifically or from any structure that does not walk a closure chain on `Get`.

Where the cost lands. The reference pays on `Get`; the refinements pay on `Put`. At put $= 50\%$:

implementation	T_{get} (μs)	T_{put} (μs)
IDS suite reference (function-as-map)	24.83	15.40
IDS (assoc-list)	10.81	16.81
balanced-tree alternative	11.40	21.12

Regressing `Get` latency against closure depth (number of `Puts` preceding a `Get` in the same worker trace) confirms the reference's chain walk: slope $0.21\,\mu$s/layer at put $= 20\%$ ($R^2 = 0.95$, Fig. 9).

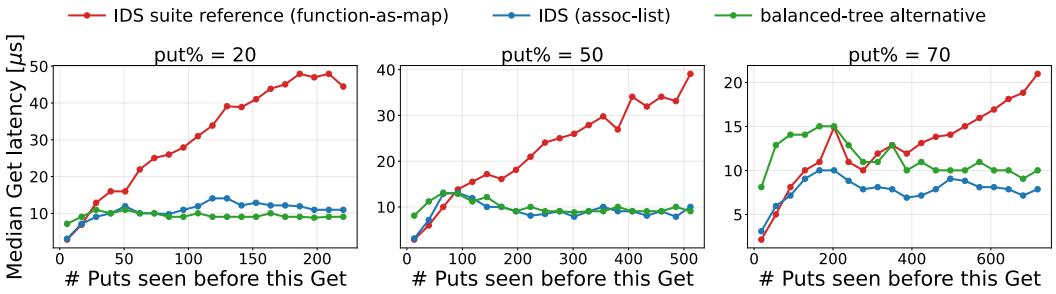


Figure 9: `Get` latency vs. closure depth (number of `Puts` preceding each `Get` in the same worker trace) for the three Monotonic Reads implementations at three put rates. The IDS suite reference's `Get` latency grows linearly with depth (slope 0.21 μs per closure layer at put $= 20\%$, $R^2 = 0.95$). IDS' assoc-list slope is an order of magnitude smaller; the balanced-tree variant's slope is within statistical noise at every put rate.

Table 9: Pass counts on verification benchmarks. Bold = best per row. The first seven are proof-only (impl and spec given); on these, IDS reduces to its DSA component since the ISA has no design space to search. The last three are joint code-and-proof tasks. Per-benchmark slices, prior-SOTA references, and CloverBench note in App. E.

Benchmark	Tasks	Prior SOTA	Codex	Claude Code	**IDS** (ours)
DafnyBench	100	52	78	76	**88**
miniCodeProps	100	49	80	86	**100**
Verus-Bench	150	137	148	148	**149**
CoqStoq	100	28	46	51	**97**
VerKer	60	18	22	26	**39**
VeriSoftBench	500	205	380	401	**432**
VeruSAGE-Bench	849	797	801	805	**838**
VERINA	189	38	136	149	**176**
AlgoVeri	77	31	51	56	**65**
CloverBench	62	–	**62**	**62**	**62**

Both refinements stay flat across every put rate measured ($|\text{slope}| < 0.025$ μs/layer): bounded data structure, no chain to walk. The balanced tree's extra Put cost comes from rebalance.

N-scaling at put $= 50\%$. The reference's throughput drops as ops-per-worker N grows; IDS stays nearly flat:

N	1,000	2,000	5,000	20,000
IDS suite reference (function-as-map)	108,000	100,000	90,000	80,000
IDS (assoc-list)	130,000	124,000	117,000	112,000

At $N = 1{,}000$ IDS is $1.20\times$ faster; at $N = 20{,}000$ the gap widens to $1.40\times$. Throughput drop from $N = 1{,}000$ to $N = 20{,}000$: reference $1.35\times$, IDS $1.16\times$. The reference's per-`Get` closure walk grows linearly with prior `Puts`, so its per-op cost rises with N; the assoc-list's lookup is bounded by the number of distinct keys in the workload (50 here), so its per-op cost is N-independent and only message-bus pressure causes the mild decline.

E Results on Proof, Annotation, and Code-and-Proof Benchmarks

IDS beats every prior method on the seven proof and annotation benchmarks, and reaches **176/189** on VERINA against prior SOTA 38 and agentic baselines Codex 136, Claude Code 149 (Table 9).

E.1 Proof and annotation benchmarks

Setup. The model receives a complete task specification (signature, preconditions, postconditions, and code where applicable) and must produce verifier-acceptable output: proof annotations on DafnyBench [35] and Verus-Bench [68], or a complete proof on miniCodeProps [34], CoqStoq [56], VerKer [58], VeriSoftBench [67], and VeruSAGE-Bench [69] (modality in Table 10). For each benchmark we use the slice the prior SOTA method published its number on:

- **DafnyBench**: 100 hardest tasks; prior SOTA is DafnyPro [7] (from the DafnyBench paper).
- **miniCodeProps**: 100 tasks from the sorting split; prior SOTA is COPRA [55].
- **Verus-Bench**: full 150 tasks; prior SOTA is AutoVerus [68].
- **CoqStoq**: 100 hardest tasks; prior SOTA is Rango [56].
- **VerKer**: 60 Linux-kernel tasks from AutoRocq's case study, scored with AutoRocq's own unmodified script; prior SOTA is AutoRocq's strongest variant (using CoqHammer) [58].
- **VeriSoftBench**: full 500 Lean 4 proof obligations from 23 formal-methods repositories, with the same 15-minute budget per task for IDS, Codex, and Claude Code (IDS: about $569 in total); prior SOTA is non-agentic multi-pass sampling with Gemini 3 Pro, which does not report its cost. We use VeriSoftBench rather than PutnamBench [57] or LeanEval [27], which target mathematical theorem proving.
- **VeruSAGE-Bench**: full 849 tasks from eight real-world Verus projects, each with the target proof function body removed; prior SOTA is Opus 4.5.

Result. IDS reaches $88/100$ on DafnyBench, $100/100$ on miniCodeProps, $149/150$ on Verus-Bench, $97/100$ on CoqStoq, $39/60$ on VerKer, $432/500$ on VeriSoftBench, and $838/849$ on VeruSAGE-Bench (Table 9), beating the strongest prior on all seven. The largest absolute gap is on CoqStoq ($+69\%$ over Rango); IDS hits a perfect $100/100$ on miniCodeProps, and on Verus-Bench all three LLM methods are within 2% of saturation (IDS 149, Codex 148, Claude Code 148 of 150). VeruSAGE-Bench is also near saturation (Sonnet 4.5 reaches 81%, Opus 4.5 93.9%), yet IDS still leads with 98.7%. On VeriSoftBench, IDS substantially improves on the non-agentic prior SOTA (432 vs. 205 of 500). Failures reflect per-task budget caps and proofs requiring tactics or library-specific lemmas the agent did not discover within budget (App. F.1).

E.2 Code-and-proof benchmarks

Setup. A second, recent line of benchmarks evaluates joint code-and-proof synthesis on single-function algorithmic tasks (sorting, search, list manipulation, basic arithmetic like factorial/gcd/Fibonacci): the model receives a natural-language description, signature, and spec, and produces both the code and a proof of correctness. These tasks are strictly simpler than the distributed-system specs of §5.2: no concurrency, no replication, no multi-step operational semantics.

- **VERINA** [73]: 189 Lean tasks; prior SOTA is iterative refinement with Lean compiler feedback (64 rounds), reported in the VERINA paper.
- **AlgoVeri** [74]: 77 problems, each expressed in Dafny, Verus, and Lean; prior SOTA $31/77$ (Gemini 3 Flash, Dafny track, reported in the AlgoVeri paper).
- **CloverBench** [53]: 62 Dafny tasks. We omit a Prior-SOTA cell because Clover (the only prior published method) evaluates a different task: its consistency-checker filters pre-existing correct triples (an 87% acceptance rate, with 13% false negatives where it rejects a known-correct triple). Our column reports end-to-end synthesis: given the spec, the model generates code and annotations, and the Dafny verifier accepts.

Result. IDS reaches $\mathbf{176/189}$ on VERINA, well above prior SOTA 38 and agentic baselines Codex 136, Claude Code 149; $\mathbf{65/77}$ on AlgoVeri's Dafny track, $2\times$ the published Gemini 3 Flash baseline of $31/77$; and all three LLM methods saturate at $62/62$ on CloverBench. Across DafnyBench, miniCodeProps, Verus-Bench, CoqStoq, VERINA, AlgoVeri, and CloverBench, 41 of the 778 tasks hit the per-task cap: 12 on DafnyBench, 13 on VERINA, 12 on AlgoVeri, 3 on CoqStoq, and 1 on Verus-Bench.

Table 10: What each cross-language benchmark gives the model and what the model must produce. ✓ = given as input; ✗ = the model must generate it. "Spec" covers preconditions and postconditions (or the theorem statement, where applicable). For DafnyBench the implementation body is given; only Dafny verification *hints* (loop invariants, assertions, decreases clauses) are stripped. miniCodeProps, VeriSoftBench, CoqStoq, and VerKer give the spec as a Lean / Rocq theorem statement.

Benchmark	Language	Verifier	Given to model				Task
			NL desc.	Signature	Spec	Code	
DafnyBench [35]	Dafny	Dafny	✗	✓	✓	✓	annotations
miniCodeProps [34]	Lean 4	Lean	✗	✓	✓	✓	proof
Verus-Bench [68]	Rust	Verus	✗	✓	✓	✓	annotations
CoqStoq [56]	Rocq	Rocq	✗	✓	✓	✓	proof
VerKer [58]	Rocq	Rocq	✗	✓	✓	✓	proof
VeriSoftBench [67]	Lean 4	Lean	✗	✓	✓	✓	proof
VeruSAGE-Bench [69]	Rust	Verus	✗	✓	✓	✓	proof
CloverBench [53]	Dafny	Dafny	✓	✓	✓	✗	code + annot.
AlgoVeri [74]	Dafny	Dafny	✓	✓	✓	✗	**code + proof**
VERINA [73]	Lean 4	Lean	✓	✓	✓	✗	**code + proof**

E.3 Distributed system: IronFleet lock

IronFleet [18] is a suite of verified distributed systems in Dafny, built over roughly 3.7 person-years. We take its lock and remove all implementations and proofs (lemmas, asserts, loop invariants), keeping the spec unmodified. IDS regenerates and proves all 15 files with no axioms or assumptions.

F IDS Architecture Details

This appendix expands on the components introduced in §4 and ablated in §5.4. The architecture instantiates the two-level CEGIS-style refinement of §4. The inner loop is the DSA (App. F.2): forward moves *Define*, *Prove*, *Decompose* extend the partial (code, proof) candidate, and the structured Rocq diagnostic from a failed step (App. F.6) is a concrete witness that the candidate cannot close as written, used by the agent to choose *Repair* or *Revert*. The outer loop is the ISA: a DSA's stalled proof attempts and bench failures (App. F.7) drive the proposer (App. F.4) on tactical stalls and the reloader (App. F.5) on strategic dead-ends. The audit step (App. F.3) is orthogonal to refinement: it is a deterministic safety gate, not a counterexample signal.

F.1 Proof methodology

Each spec is a Rocq `Module Type` carrying a `State` type, an `Update` type, seven method signatures (`getReq`, `getGuard`, `get`, `getRes`, `putReq`, `putGuard`, `put`), and an operational semantics on client–replica configurations. The implementation IDS produces is an `AlgDef` module providing the same seven methods at concrete state types, plus a `Refinement` module proving forward simulation: every step the implementation can take corresponds to a step the abstract spec can take, on related states. From the `Refinement` we obtain a `TraceInclusion` theorem: every observable trace of the implementation is also a trace of the abstract spec.

The tactic vocabulary is standard Rocq: `inversion`, `destruct`, `subst`, `rewrite`, `apply`, `econstructor; eauto`, `simpl`, `lia`, `congruence`, with case-analysis on `Nat.eq_dec` for equality decisions on identifiers and timestamps. No `firstorder`/`intuition` shortcuts; no `Admitted`, no `Axiom`, no `Hypothesis`, no `Parameter`. Closure is verified by Rocq's kernel.

F.2 DSA: deductive synthesis

The DSA is a Codex (or Claude Code) session per workspace performing the deductive-synthesis loop of §4. At each search-tree node the agent moves forward with one of three primitives:

- *Define*: declare or extend a state/message type or function body, with `admit` placeholders for parts not yet filled in (e.g. adding a `Cell` record to the per-key store with `cell_deps` stubbed).
- *Prove*: close an open proof obligation by replacing `Admitted` with a tactic sequence ending in `Qed`, or extend an existing tactic block.
- *Decompose*: state a helper lemma as `Admitted` and use it to advance the proof of an existing lemma; the helper is discharged in a later iteration.

After each forward step `make` runs and Rocq's type-checker is the only oracle. The definitions and lemmas the proof depends on stay compiled in separate files, so `make` recompiles only the working file the step edits: about 4 seconds on Chapar, versus 12–17 seconds for a full build. On error, the agent backtracks with one of two primitives:

- *Repair*: re-attempt the failed step with a different tactic (e.g. `induction` instead of `destruct`; `lia` instead of `omega`).
- *Revert*: rewind to an earlier search-tree node and choose a different forward primitive (typically when the same compile error recurs three or more times).

The agent's initial brief is read from `CLAUDE.md` in the workspace and instantiates the DSA of §4 in four briefings (methodology, auditability, design, tactical playbook). The brief shown below is for the Rocq setting (our IDS suite specs and Chapar CC, plus CoqStoq and VerKer); for cross-language benchmarks we swap the verifier and surface tools to match the target language: Dafny (DafnyBench, AlgoVeri-Dafny, CloverBench, IronFleet lock), Verus (Verus-Bench, VeruSAGE-Bench), and Lean 4 (VERINA, miniCodeProps, VeriSoftBench), with success criterion adapted accordingly (e.g. `dafny verify` accepts, `lake build` succeeds, `verus` reports zero errors).

Role: Deductive Synthesis Agent. Given a specification, incrementally produce an implementation and a machine-checked proof that it satisfies the specification. The type-checker is the only oracle and accepts a partial state (with deferred holes) as a valid intermediate.
Schema. Synthesis is a finite sequence of well-typed steps; the file must type-check (possibly with deferred holes) after every step. Take one forward move:

- *Define* (refinement): introduce or extend a `Definition`, `Record`, `Inductive`, `Fixpoint`, or `Module`; bodies may contain `admit` or invoke `Admitted` lemmas.
- *Prove* (closure): replace an `Admitted` lemma with a tactic sequence ending in `Qed`, or extend an existing partial proof.
- *Decompose* (lemma introduction): state a helper lemma as `Admitted`, use it to advance the current proof, and discharge it later. This is the deferred-hole device of deductive synthesis.

On error, backtrack: *Repair* (different tactic on the same goal, e.g. `induction` instead of `destruct`, `lia` instead of `omega`) or *Revert* (rewind to an earlier state and try a different forward move; trigger when the same error recurs three or more times).
Invariants. (I1) The file type-checks after every step. (I2) Every `Admitted` is a deferred hole, not a permanent assumption: all `Admitted` must be closed before reporting *done*. (I3) The specification is fixed. (I4) No `Axiom`, `Hypothesis`, `Parameter`, or unguarded recursion.
Design. State types should be concrete (records, bounded lists, indexed nats), not function types whose `override` builds closure chains under repeated update. State must not grow unboundedly with the number of operations: traces, logs, and accumulated dependency lists belong in the proof context, not the runtime state.
Termination. Report *done* when (a) the file type-checks with zero `Admitted` and zero `admit`, and (b) the audit step passes (no vacuous proofs; no specification edits; non-trivial method bodies). The coordinator additionally extracts the implementation and runs the benchmark harness; if extraction or execution fails, the run reverts.
Tactical playbook.

- Build simple to complex: prove small structural facts first, then combine.
- When stuck on a subgoal, read it carefully; the goal often names the missing helper.
- Use `admit` (lowercase) to skip subgoals temporarily; the final file must have zero `Admitted` and zero `admit`.

The −G ablation (§5.4) removes the tactical playbook, that is, the tactic and decomposition guidance, keeping the task constraints and the Rocq feedback loop. The playbook is generic proof-engineering advice, not domain-specific algorithms, state representations, or proof content; with only tactics adapted per language, it applies across Dafny, Lean, Verus, and Rocq.

App. G walks through an end-to-end illustration of this schema on a small target (`all_less_than`): each of the five synthesis steps is annotated with its move (*Define*, *Prove*, *Decompose*), with every intermediate listing type-checking under the kernel.

F.3 Audit step

The audit step is a deterministic Python script that gates closure with kernel-level verdicts (no model outputs). It encapsulates the cheating patterns we have observed during runs.

Static checks.

- `Admitted` count is zero.
- After stripping comments, `Axiom`, `Hypothesis`, and `Parameter` counts are zero.
- `make clean && make` exits zero.
- The specification is unchanged.

Vacuous-proof patterns. The script also matches the closed file against five patterns:

- A `put` that returns a fixed constant the spec accepts on every input.
- An always-true `getGuard` that admits any read.
- An always-false `putGuard` under which the theorem holds trivially.
- A forward-simulation theorem stated over an empty domain so that it holds vacuously.
- A state-only stub whose `Update` drops the payload and returns the initial state on every operation.

A candidate that fails any check is sent to the ISA; only candidates passing all checks count as audit-clean closures.

F.4 Proposer (ISA tactical role)

After the coordinator counts no progress across 10 consecutive polls, the proposer fires as a one-shot LLM session. It receives the work file, every `Admitted` lemma's surrounding 17-line context, and the most recent `make` error with 4 lines of trailing context when present, and writes its output to `GUIDANCE.md` in the workspace.

Role. You are an expert Rocq proof advisor. Read the work file and focus on the proof goals and error context below. Do NOT edit the `.v` file.
For each Admitted proof, follow this checklist:

1. **Read the lemma statement.** What does it claim? What are the hypotheses? What is the conclusion type?
2. **Identify the proof structure needed:** `forall x, P x` → `intros x`; inductive type (list, trace, nat) → `induction` on it; step/transition → `destruct` on the step type, then case analysis; simulation → invariant + induction on trace length; two functions equal → `functional_extensionality`.
3. **Identify missing helper lemmas.** For each subgoal you cannot close directly, state the EXACT helper (name, signature, informal meaning). Suggest: prove it separately; `Admit` it in the main proof; close it.
4. **If the proof is > 30 lines, decompose:** by cases (`X_case_C1`, `X_case_C2`, ... for each constructor); by subgoal (if $A \wedge B$, prove A and B separately); extract common patterns repeating 3+ times.
5. **Suggest specific tactic sequences,** not vague directions.

For compile errors, diagnose using a decision tree (e.g., *"Cannot unify X with Y"* → type mismatch, use `Check expr.`; *"Tactic failure"* → try `lia` instead of `omega`; *"No matching clauses"* → incomplete pattern match).
Output. Write `GUIDANCE.md` with one section per Admitted proof or error: (a) the diagnosis, (b) the exact tactic sequence to try, (c) any helper-lemma signatures.

On Chapar, the proposer chose three sub-lemma classes (per-message, per-store-entry, per-replica-clock) by recognizing that the global delivery invariant decomposes cleanly along these axes, none of which Rocq's automation closes alone.

F.5 Reloader (ISA strategic role)

The reloader escalates every 20 polling cycles when no progress is detected: Level 1 at stall cycle 20, Level 2 at 40, Level 3 at 60, and so on. Each level is a one-shot LLM session that reads `DESIGN_LOG.md`, `STATUS.md`, and every prior level's output, and writes `META_GUIDANCE_L{N}.md`. The coordinator consumes the output to rewrite its plan and spawn a fresh worker on the new design.

Role. You are a level-N formal verification architect. Read `DESIGN_LOG.md`, `STATUS.md`, and every prior `META_GUIDANCE_L{1..N-1}.md`: the history of all approaches tried and why each failed.
Synthesize a NEW approach that avoids ALL recorded failures. Think across these dimensions:

1. **Data representation.** Concrete types for state and messages: records, bounded lists, nats, not functions.
2. **Abstraction function.** How concrete state maps to abstract state. This is the KEY creative choice; if previous attempts failed on the simulation proof, the abstraction was wrong.
3. **Invariant.** What property holds at every reachable state. The invariant must be (a) true initially, (b) preserved by every step, (c) strong enough to prove the final theorem.
4. **Proof decomposition.** Which helper lemmas are needed, stated explicitly with signatures.
5. **Minimum information.** For each function that takes a collection, could the caller pass a smaller collection and still get a correct result? The most precise design passes only what is needed per call.

Output. Write `META_GUIDANCE_L{N}.md` as a CONCRETE blueprint a worker can directly implement: exact `Record` types, method signatures, abstraction function, invariant, and lemma decomposition. Do NOT write vague advice; do NOT spawn workers.

On Chapar, the reloader chose a per-key store entry recording sender, sender clock, and dependency vector; alternative layouts (e.g. a single global per-replica state) leave the delivery lemma unprovable from local information.

F.6 Rocq feedback

Each `make` invocation captures stdout/stderr; on non-zero exit, the coordinator extracts every `Error:` line plus 2 lines of preceding context and 4 lines of trailing context as a single error block. The block is included verbatim in the worker's next prompt alongside the proof goal, the local hypothesis context, and the tactic backtrace where applicable. No normalization or summarization is applied: the structured Rocq diagnostic carries the localization signal. Replacing this stream with a binary accept/reject collapses the search (§5.4 reports $\leq 1/3$ closure on every Rocq spec under $-$VF).

F.7 Performance feedback

When a worker reaches a verifier-clean state with a design hash distinct from prior runs, the coordinator invokes the benchmark harness asynchronously. The harness extracts the implementation to OCaml, deploys it on the cluster (App. D), and writes results to `PERF_RESULTS.md` in the workspace. Throughput targets are calibrated per-spec from the published reference; implementations whose throughput falls below the target are flagged and trigger the reloader on the next escalation.

G Synthesis Progression: `all_less_than`

This appendix walks through one full progression of IDS' deductive-synthesis loop on a small target. Step 0 and the Final State are complete Rocq files that compile; the listings for intermediate Steps 1–5 show only the parts that change from the previous step, with not-yet-filled bodies stubbed as `Parameter`. These stubs are only for exposition: in IDS runs, Invariant I4 forbids `Parameter`, and the DSA leaves unfilled bodies as `admit` or `Admitted` (App. F.2). The progression on real distributed-system specs follows the same pattern at much larger scale (§5.2); the small target keeps each listing short enough to read end-to-end.

The five synthesis steps below are an instance of the deductive-synthesis schema of App. F.2. Step 1 is a *Decompose* (split the goal into a `nil` and a `cons` helper lemma). Step 2 combines *Define* (fill the `nil` body) with *Prove* (close the `nil` helper). Step 3 is another *Decompose* (split the `cons` helper on the head). Step 4 combines *Define* (the recursive call) with *Prove* (close the `then` helper). Step 5 combines *Define* (fill the `else` branch) with *Prove* (close the `else` helper). The file type-checks

at every step (Invariant I1); `Admitted` placeholders carry the deferred holes forward and are all discharged by the Final state.

The target is `all_less_than : list nat -> nat -> bool`, returning `true` iff every element of the input list is strictly less than the bound `n`. The correctness statement is the iff-equivalence with `Forall`:

$$\texttt{all_less_than l n = true} \Leftrightarrow \texttt{Forall (fun x => x < n) l.}$$

G.1 Step 0: bare specification

The implementation is a `Parameter`; the correctness lemma is `Admitted`. The file type-checks but proves nothing.

```
Require Import Coq.Lists.List.

Parameter all_less_than : list nat -> nat -> bool.

Lemma all_less_than_correct : forall (l : list nat) (n : nat),
  all_less_than l n = true <-> Forall (fun x => x < n) l.
Proof.
Admitted.
```

Listing 1: Step 0: spec only.

G.2 Step 1: case-split on the input list

The agent commits to recursing on `l`. The implementation becomes a `Definition` with a `match` on `nil`/`cons`; the branch bodies are stubbed as `Parameters`. The correctness lemma is split into a `_nil` helper and a `_cons` helper, both `Admitted`; the main lemma now closes by `induction` on `l`, applying each helper at its leaf.

```
Parameter nil_body : bool.
Parameter cons_body : nat -> list nat -> bool.

Definition all_less_than (l : list nat) (n : nat) : bool :=
  match l with
  | nil => nil_body
  | x :: xs => cons_body x xs
  end.

Lemma all_less_than_correct_nil : forall (n : nat),
  all_less_than nil n = true <-> Forall (fun x => x < n) nil.
Proof. Admitted.

Lemma all_less_than_correct_cons :
  forall (x : nat) (xs : list nat) (n : nat),
  all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs).
Proof. Admitted.

Lemma all_less_than_correct : forall (l : list nat) (n : nat),
  all_less_than l n = true <-> Forall (fun x => x < n) l.
Proof.
  intros l n. induction l as [| x xs IH].
  - apply all_less_than_correct_nil.
  - apply all_less_than_correct_cons.
Qed.
```

G.3 Step 2: fill the `nil` branch and close the `nil` helper

The agent fills `nil_body` with `true` (replacing the `Parameter`) and proves `_nil`: both sides of the iff are vacuously satisfied (`true = true`; `Forall _ nil` holds by the empty constructor). The `cons` body is still a `Parameter`.

```
Definition all_less_than (l : list nat) (n : nat) : bool :=
  match l with
  | nil => true
  | x :: xs => cons_body x xs
  end.

Lemma all_less_than_correct_nil : forall (n : nat),
  all_less_than nil n = true <-> Forall (fun x => x < n) nil.
Proof. intros. simpl. split; intros; constructor. Qed.
```

G.4 Step 3: introduce the `if` branch and split the `cons` helper

The agent commits to comparing `x` against `n` via `x <? n`, leaving both branches as `Parameter` stubs. The `_cons` helper is split into a `_then` sub-helper (under `x < n`) and an `_else` sub-helper (under $\neg$`(x < n)`), and `_cons` now proves itself by case-splitting on `Nat.ltb_spec` and applying the appropriate sub-helper. The main lemma now passes the induction hypothesis to `_cons`.

```
Parameter then_body else_body : nat -> list nat -> nat -> bool.

Definition all_less_than (l : list nat) (n : nat) : bool :=
  match l with
  | nil => true
  | x :: xs => if x <? n then then_body x xs n else else_body x xs n
  end.

Lemma all_less_than_correct_cons_then :
  forall (x : nat) (xs : list nat) (n : nat),
  x < n -> (all_less_than xs n = true <-> Forall (fun x => x < n) xs) ->
  (all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs)).
Proof. Admitted.

Lemma all_less_than_correct_cons_else :
  forall (x : nat) (xs : list nat) (n : nat),
  ∼(x < n) -> (all_less_than xs n = true <-> Forall (fun x => x < n) xs) ->
  (all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs)).
Proof. Admitted.

Lemma all_less_than_correct_cons :
  forall (x : nat) (xs : list nat) (n : nat),
  all_less_than xs n = true <-> Forall (fun x => x < n) xs ->
  all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs).
Proof.
  intros. destruct (Nat.ltb_spec x n) as [Hlt | Hge].
  - apply all_less_than_correct_cons_then; assumption.
  - apply all_less_than_correct_cons_else; [apply Nat.le_ngt; assumption |
      assumption].
Qed.

Lemma all_less_than_correct : forall (l : list nat) (n : nat),
  all_less_than l n = true <-> Forall (fun x => x < n) l.
Proof.
  intros l n. induction l as [| x xs IH].
  - apply all_less_than_correct_nil.
  - apply all_less_than_correct_cons. assumption.
Qed.
```

G.5 Step 4: fill the `then` branch with the recursive call

The agent fills `then_body` with `all_less_than xs n`: the implementation now actually recurses, so `Definition` is promoted to `Fixpoint`. The `_then` helper proves cleanly: under `x < n`, applying

`Nat.ltb_lt` rewrites the `if`-condition to `true`, and `Forall` on `x :: xs` reduces to `x < n` on the head plus `Forall` on `xs`, both available.

```
Fixpoint all_less_than (l : list nat) (n : nat) : bool :=
  match l with
  | nil => true
  | x :: xs => if x <? n then all_less_than xs n else else_body x xs n
  end.

Lemma all_less_than_correct_cons_then :
  forall (x : nat) (xs : list nat) (n : nat),
  x < n -> (all_less_than xs n = true <-> Forall (fun x => x < n) xs) ->
  (all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs)).
Proof.
  intros x xs n Hlt H_ind. simpl.
  apply Nat.ltb_lt in Hlt. rewrite Hlt.
  split; intros.
  - constructor; [apply Nat.ltb_lt; assumption | tauto].
  - inversion H. tauto.
Qed.
```

G.6 Step 5: fill the `else` branch and close the `else` helper

The agent's first attempt is to recurse: `else_body := all_less_than xs n` ("skip the out-of-bound element, keep checking"). The implementation type-checks, but the `_else` obligation does not: under $\neg(x < n)$ the goal becomes `all_less_than xs n = true` $\Leftrightarrow$ `Forall (fun y => y < n) (x :: xs)`, which fails on `xs = nil`, `x = 5`, `n = 3` (LHS is `true`; RHS demands `5 < 3`). Rocq rejects the proof. The agent backtracks and replaces the `else` body with `false`: the function now returns false on any out-of-bound element, the `_else` helper closes (LHS becomes `false = true`, closed by `discriminate`; RHS gives `x < n` from inversion of `Forall`, contradicting $\neg(x < n)$).

```
(* Attempt 1: else_body := all_less_than xs n (rejected: iff fails) *)
(* Attempt 2: else_body := false (accepted) *)

Lemma all_less_than_correct_cons_else :
  forall (x : nat) (xs : list nat) (n : nat),
  ~(x < n) -> (all_less_than xs n = true <-> Forall (fun x => x < n) xs) ->
  (all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs)).
Proof.
  intros x xs n Hge _. simpl.
  apply Nat.le_ngt in Hge. apply Nat.ltb_ge in Hge. rewrite Hge.
  split.
  - discriminate.
  - intros HF. inversion HF; subst.
    apply Nat.ltb_lt in H1. congruence.
Qed.
```

G.7 Final state

After five steps every `Parameter` stub and `Admitted` placeholder is closed. Rocq's kernel accepts the file, verifying `all_less_than_correct` on every input.

```
Require Import Coq.Lists.List.
Require Import Coq.Arith.PeanoNat.

Fixpoint all_less_than (l : list nat) (n : nat) : bool :=
  match l with
  | nil => true
  | x :: xs => if x <? n then all_less_than xs n else false
  end.

```

```
Lemma all_less_than_correct_nil : forall (n : nat),
  all_less_than nil n = true <-> Forall (fun x => x < n) nil.
Proof. intros. simpl. split; intros; constructor. Qed.

Lemma all_less_than_correct_cons_then :
  forall (x : nat) (xs : list nat) (n : nat),
  x < n -> (all_less_than xs n = true <-> Forall (fun x => x < n) xs) ->
  (all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs)).
Proof.
  intros x xs n Hlt H_ind. simpl.
  apply Nat.ltb_lt in Hlt. rewrite Hlt.
  split; intros.
  - constructor; [apply Nat.ltb_lt; assumption | tauto].
  - inversion H. tauto.
Qed.

Lemma all_less_than_correct_cons_else :
  forall (x : nat) (xs : list nat) (n : nat),
  ~(x < n) -> (all_less_than xs n = true <-> Forall (fun x => x < n) xs) ->
  (all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs)).
Proof.
  intros x xs n Hge _. simpl.
  apply Nat.le_ngt in Hge. apply Nat.ltb_ge in Hge. rewrite Hge.
  split.
  - discriminate.
  - intros HF. inversion HF; subst.
    apply Nat.ltb_lt in H1. congruence.
Qed.

Lemma all_less_than_correct_cons :
  forall (x : nat) (xs : list nat) (n : nat),
  all_less_than xs n = true <-> Forall (fun x => x < n) xs ->
  all_less_than (x :: xs) n = true <-> Forall (fun x => x < n) (x :: xs).
Proof.
  intros. destruct (Nat.ltb_spec x n) as [Hlt | Hge].
  - apply all_less_than_correct_cons_then; assumption.
  - apply all_less_than_correct_cons_else; [apply Nat.le_ngt; assumption |
      assumption].
Qed.

Lemma all_less_than_correct : forall (l : list nat) (n : nat),
  all_less_than l n = true <-> Forall (fun x => x < n) l.
Proof.
  intros l n. induction l as [| x xs IH].
  - apply all_less_than_correct_nil.
  - apply all_less_than_correct_cons. assumption.
Qed.
```

Listing 2: Step 5 (final): implementation and proof, fully verified.

The progression on real distributed-system specs follows the same pattern at much larger scale: dozens of helper lemmas, hundreds of lines of implementation, and tactics that range over inductive simulation arguments rather than single-step rewrites. §4 describes how IDS drives this process; the case-study trajectory in §5 shows it on the causal-consistency specification.

H Limitations, Broader Impacts, and Ethics

The main paper presents IDS' design and evaluation; this appendix addresses the limitations of the work, the broader impacts of verified-synthesis tooling, and ethical considerations.

Limitations. IDS requires a formal specification as input; we do not synthesize specifications from natural language; automating specification development is an orthogonal open problem (§6). Compute cost is 2–11 hours and $52–$155 per closed spec, substantially below the months-to-years of

expert proof effort it replaces but still meaningful. We evaluate system-level synthesis on distributed key-value-store consistency and the IronFleet lock, and proof synthesis alone also on the Linux kernel (VerKer) and on Rust systems such as OS kernels, memory allocators, and serialization code (VeruSAGE-Bench); generality of the former to other verified-synthesis domains (compilers, OS kernels, cryptographic protocols) is conjectured, not measured. IronFleet's larger IronRSL and IronKV are future work. For the key-value stores, IDS verifies at the protocol level: Rocq checks the implementation and its refinement proof. As in Verdi [65] and Chapar [31], the trusted computing base is the Rocq kernel, the extraction, the OCaml toolchain, the distributed runtime (§5.1 and App. D), and the hardware. Our synthesis method is independent of this boundary; extending down the stack, via CertiCoq [3] for verified compilation, together with validation of the runtime, is a promising direction.

Broader impacts. Distributed systems are common in production software (databases, message queues, cloud services), but formal verification of these systems has remained out of reach for most production code because hand-written proofs take person-years of expert effort. IDS reduces that effort to hours of compute, putting verified distributed software within reach of production teams. Kernel-checked proofs make IDS' outputs safer than unverified LLM-generated code: the verifier rules out silent bugs against the specification. As with all formal verification, mis-specification remains a residual risk.

Ethics. The work involves no human subjects and no scraped data; the verified outputs introduce no dual-use risks beyond those already present in unverified LLM code generation.

I IDS Suite

IDS suite is the benchmark dataset of six distributed key-value-store specifications we release alongside the system; together with Chapar's published causal-consistency spec [31], they form the seven specifications evaluated in §5. Each specification fixes (i) the abstract spec state and operations the application sees, (ii) the concrete replica state and operations the implementation runs, and (iii) the consistency property the implementation must satisfy. This appendix gives the formal definition of every spec, plus the reference implementations we benchmark in §5. The framework setup below introduces the notation: client programs (Fig. 10), the seven-method `AlgDef` interface (Fig. 11), the configuration syntax (Fig. 12), the concurrent operational semantics (Fig. 13), the refinement hierarchy (Fig. 14), and a relaxed baseline spec (Fig. 15).

Summary. Table 11 lists the six IDS suite specifications plus Chapar's published CC. Each is then introduced in plain English and immediately followed by its formal definition and reference implementation.

Table 11: The six IDS suite specifications, plus Chapar's published CC for reference.

spec	abbreviation	informal guarantee
Read-Your-Writes	RYW	A client's read sees that client's own most recent write.
Monotonic Reads	MR	A client's reads never go back in time relative to writes the client has already seen.
Monotonic Writes	MW	Writes from one client are applied in order at every replica.
Read-Your-Writes + Monotonic Writes	RYW+MW	Composition of RYW and MW; an implementation must refine both.
Causal Consistency	CC	A read returns a value whose causal predecessors have all been observed at the responding replica.
Labeled Causal Consistency	LCC	CC parameterized by per-message topic labels: each client subscribes to a subset of labels and only sees causality enforced on those.
Chapar Causal Consistency	Chapar CC	Chapar's published CC specification [31]; see the original paper for its formal definition.

k	:	K	Key
v	:	$V \supseteq K$	Value
x	:	V	Variable
i			Unique Identifiers
$s : S$	$::=$	$put^i(k, v);\ s$	Statement
	$\mid$	$x \leftarrow get^i(k);\ s$	
	$\mid$	$skip$	
	$\mid$		Extended Syntax for Internal Statements
	$\mid$	$\oslash\, x \leftarrow get^i(k);\ s$	Blocked Get
c	:	C	Clients
a	:	$A = C \mapsto S$	Application

Figure 10: Client Programs

$\mathbb{I} = (\mathsf{CState}, \mathsf{c\text{-}init}, \mathsf{RState}, \mathsf{r\text{-}init},$ $\mathsf{get\text{-}req}, \mathsf{get\text{-}guard}, \mathsf{get}, \mathsf{get\text{-}res},$ $\mathsf{put\text{-}req}, \mathsf{put\text{-}guard}, \mathsf{put})$	Implementation
C	Clients
$\mathsf{CState} : \mathsf{Type}$	Client State
$\mathsf{c\text{-}init} : C \to \mathsf{CState}$	Client Initial State
R	Replicas
$\mathsf{RState} : \mathsf{Type}$	Replica State
$\mathsf{r\text{-}init} : R \to V \to \mathsf{RState}$	Replica Initial State
$\mathsf{Get\text{-}req\text{-}payload}, \mathsf{Get\text{-}res\text{-}payload} : \mathsf{Type}$	Get Payload Types
$\mathsf{get\text{-}req} : (K)(C, \mathsf{CState}) \to (\mathsf{Get\text{-}req\text{-}payload}, \mathsf{CState})$	Get Request (at client)
$\mathsf{get\text{-}guard} : (K)(\mathsf{Get\text{-}req\text{-}payload})(C, R, \mathsf{RState}) \to \mathsf{Bool}$	Get Guard (at replica)
$\mathsf{get} : (K)(\mathsf{Get\text{-}req\text{-}payload})(C, R, \mathsf{RState}) \to (V \times \mathsf{Get\text{-}res\text{-}payload}, \mathsf{RState})$	Get (at replica)
$\mathsf{get\text{-}res} : (K, V)(\mathsf{Get\text{-}res\text{-}payload})(C, \mathsf{CState}) \to \mathsf{CState}$	Get Response (at client)
$\mathsf{Put\text{-}req\text{-}payload} : \mathsf{Type}$	Put Payload Type
$\mathsf{put\text{-}req} : (K, V)(C, \mathsf{CState}) \to (\mathsf{Put\text{-}req\text{-}payload} \times \mathsf{CState})$	Put Request (at client)
$\mathsf{put\text{-}guard} : (K, V)(\mathsf{Put\text{-}req\text{-}payload})(C, R, \mathsf{RState}) \to \mathsf{Bool}$	Put Guard (at replica)
$\mathsf{put} : (K, V)(\mathsf{Put\text{-}req\text{-}payload})(C, R, \mathsf{RState}) \to \mathsf{RState}$	Put (at replica)

Figure 11: Key-Value Store Implementation Interface

$$
\begin{array}{rcll}
\mathbb{I} & = & (\mathsf{CState}, \mathsf{c\text{-}init}, \mathsf{RState}, \mathsf{r\text{-}init}, & \text{Implementation} \\
 & & \ \mathsf{get\text{-}req}, \mathsf{get\text{-}guard}, \mathsf{get}, \mathsf{get\text{-}res}, & \\
 & & \ \mathsf{put\text{-}req}, \mathsf{put\text{-}guard}, \mathsf{put}) & \\
W : \mathcal{W} & \coloneqq & \langle \mathcal{C}, \mathcal{R}, \mathcal{N} \rangle & \text{World} \\
c & : & C & \text{Clients} \\
\sigma & : & \mathsf{CState} & \text{Client State} \\
\mathcal{C} & : & C \mapsto \langle \mathsf{CState}, S \rangle & \text{Client Programs} \\
r & : & R & \text{Replicas} \\
\varsigma & : & \mathsf{RState} & \text{Replica State} \\
\mathcal{R} & : & R \mapsto \mathsf{RState} & \text{Replica States} \\
\mathcal{N} & : & \mathbb{PM}(M) & \text{Network} \\
M & : & C \times R \times \mathit{Get\text{-}req}(I, K, \mathsf{Get\text{-}req\text{-}payload}) & \text{Message} \\
 & | & R \times C \times \mathit{Get\text{-}res}(I, K, V, \mathsf{Get\text{-}res\text{-}payload}) & \\
 & | & C \times R \times \mathit{Put\text{-}req}(K, V, \mathsf{Put\text{-}req\text{-}payload}) & \\
l & ::= & c \triangleright \mathit{get}^i(k) & \text{Label} \\
 & | & r \triangleright \mathit{get}^i(k) : v & \\
 & | & c \triangleright \mathit{get}^i(k) : v & \\
 & | & c \triangleright \mathit{put}(k, v) & \\
 & | & r \triangleright \mathit{put}(k, v) & \\
h & ::= & l^* & \text{History} \\
\mathsf{ext}(h) & ::= & h \mid \{c \triangleright \mathit{get}^i(k) : v,\ c \triangleright \mathit{put}(k, v)\} & \text{External History}
\end{array}
$$

$$
\begin{array}{rcl}
\circ & : & \mathsf{Unit} \\
\mathbb{PM}(S) & \triangleq & \text{The multiset powerset of the set } S \\
W_0(a)_{\mathbb{I}} & \triangleq & \langle \overline{[c \mapsto \langle \mathsf{c\text{-}init}(c), a(c) \rangle}_{c \in C}], [\overline{r \mapsto \mathsf{r\text{-}init}(r, v_0)}_{r \in R}], \emptyset \rangle \\
v_0 & \triangleq & \text{The initial value}
\end{array}
$$

Figure 12: The State of the Key-Value Store Operational Semantics

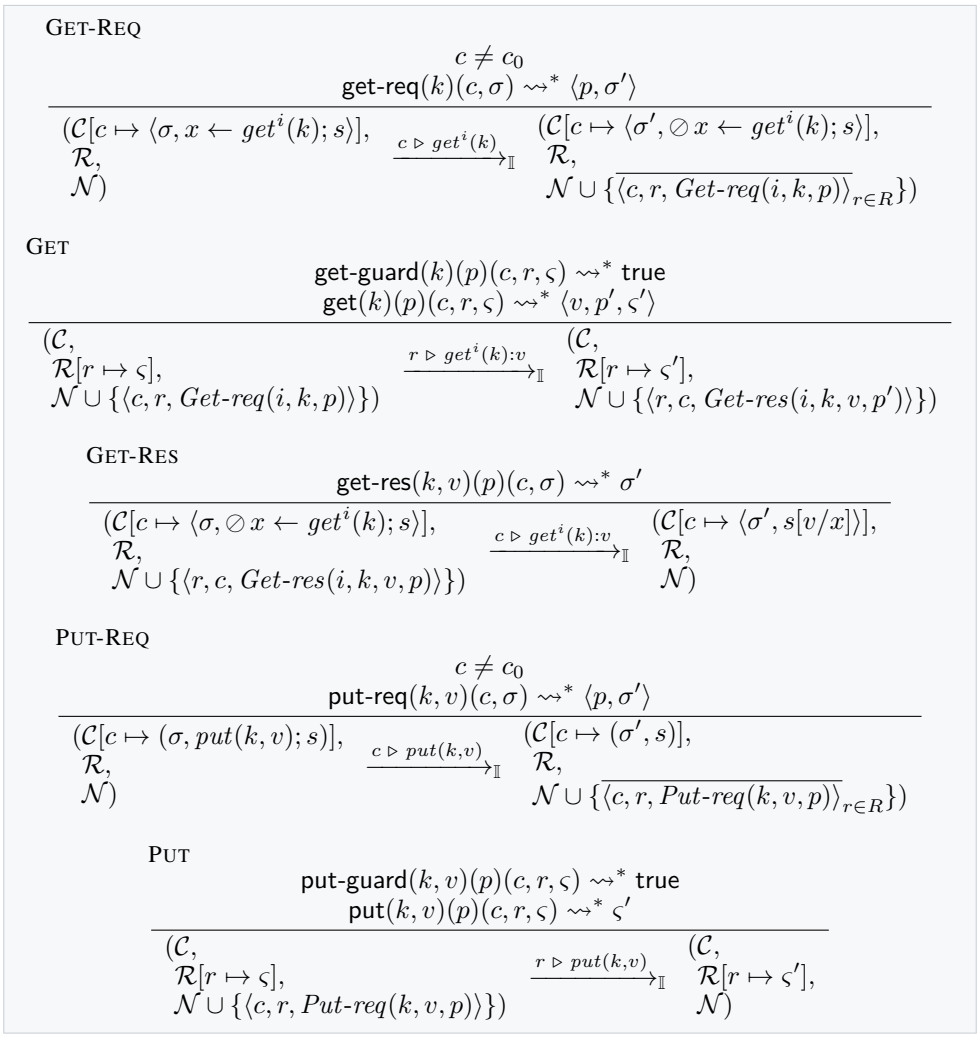
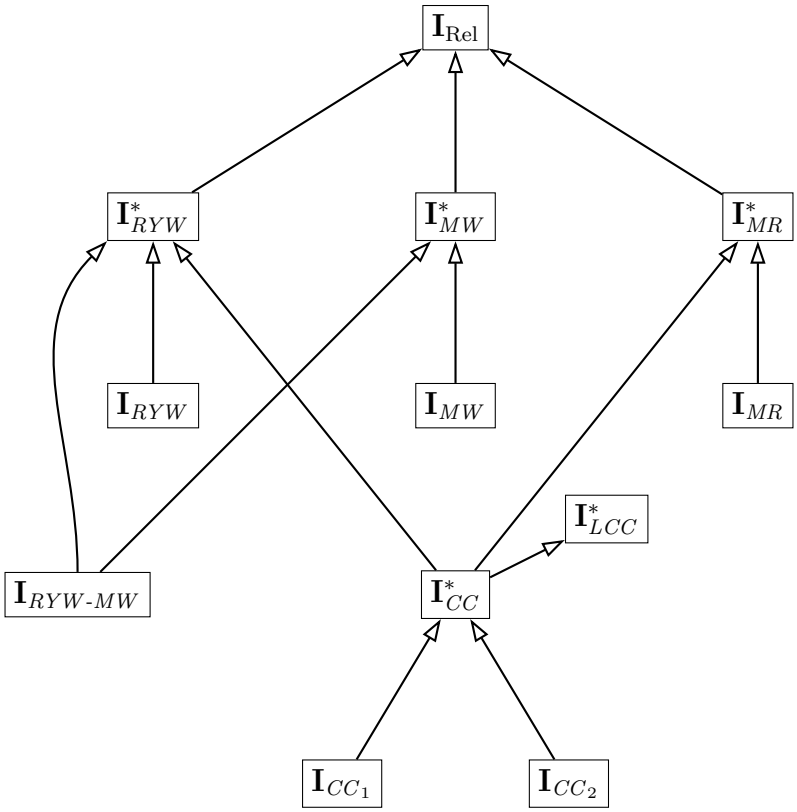


Figure 13: Key-Value Store Operational Semantics $\rightarrow_{\mathbb{I}}$ for the implementation $\mathbb{I} = (\mathsf{CState}, \mathsf{c\text{-}init}, \mathsf{RState}, \mathsf{r\text{-}init}, \mathsf{get\text{-}req}, \mathsf{get\text{-}guard}, \mathsf{get}, \mathsf{get\text{-}res}, \mathsf{put\text{-}req}, \mathsf{put\text{-}guard}, \mathsf{put})$.

The get operations are synchronous. Therefore, the semantics implicitly preserves the order of gets and succeeding gets and puts.

Definition 1 (Trace Inclusion)**.** An implementation $\mathbb{I}_2$ trace-includes another $\mathbb{I}_1$, written $\mathbb{I}_2 \sqsubseteq \mathbb{I}_1$, if for all W_2, and h_2, if $W_0(a)_{\mathbb{I}_2} \xrightarrow{h_2}{}^*_{\mathbb{I}_2} W_2$, then there exist h_1 and W_1 such that $W_0(a)_{\mathbb{I}_1} \xrightarrow{h_1}{}^*_{\mathbb{I}_1} W_1$ and $\mathsf{ext}(h_1) = \mathsf{ext}(h_2)$.

Definition 2 (Convergence)**.** An implementation $\mathbb{I}$ is convergent if for all $\mathcal{R}$, if $W_0(a)_{\mathbb{I}} \rightarrow^*_{\mathbb{I}} \langle _, \mathcal{R}, \emptyset \rangle$, then for all k, p, c, r_1, r_2 and v, if $\mathsf{get\text{-}guard}(k)(p)(c, r_1, \mathcal{R}[r_1]) \rightsquigarrow^* \mathsf{true}$ and $\mathsf{get}(k)(p)(c, r_1, \mathcal{R}[r_1]) \rightsquigarrow^* \langle v, _, _ \rangle$ then $\mathsf{get\text{-}guard}(k)(p)(c, r_2, \mathcal{R}[r_2]) \rightsquigarrow^* \mathsf{true}$ and $\mathsf{get}(k)(p)(c, r_2, \mathcal{R}[r_2]) \rightsquigarrow^* \langle v, _, _ \rangle$.



$\mathbf{I}_{\text{Rel}}$	Relaxed Specification (Fig. 15)
$\mathbf{I}^*_{RYW}$	Read-Your-Writes Specification (Fig. 16)
$\mathbf{I}_{RYW}$	Read-Your-Writes Implementation (Fig. 17)
$\mathbf{I}^*_{MR}$	Monotonic Reads Specification (Fig. 18)
$\mathbf{I}_{MR}$	Monotonic Reads Implementation (Fig. 19)
$\mathbf{I}^*_{MW}$	Monotonic Writes Specification (Fig. 20)
$\mathbf{I}_{MW}$	Monotonic Writes Implementation (Fig. 21)
$\mathbf{I}_{RYW\text{-}MW}$	Read-Your-Writes and Monotonic Writes Implementation (Fig. 22)
$\mathbf{I}^*_{LCC}$	Labeled Causal Consistency Specification (Fig. 24)
$\mathbf{I}^*_{CC}$	Causal Consistency Specification (Fig. 23)
$\mathbf{I}_{CC_1}$	Causal Consistency Implementation 1 (Fig. 25)
$\mathbf{I}_{CC_2}$	Causal Consistency Implementation 2 (Fig. 26)

Figure 14: Refinement Hierarchy

$\mathbb{I}_{Rel}$ (RELAXED SPECIFICATION)

```
CState := Unit                                              Client State
c-init(c) := ∘                                              Client Initial State
RState := K ↦ V                                             Replica State
r-init(r, v0) := [k ↦ v0]                                   Replica Initial State

Get-req-payload := Get-res-payload := Unit                  Get payload types
get-req (_)(_, σ) :=                                       At client: Get Request
  ret ⟨∘, σ⟩                                                No payload
get-guard (_)(_)(_, _, _) :=                                At replica: Get Guard
  ret true
get (k)(_)(_, _, ς) :=                                      At replica: Get
  let v ← ς(k)
  ret ⟨v, ∘, ς⟩                                             No payload
get-res (_, _)(_)(_, σ) :=                                  At client: Get Response
  ret σ

Put-req-payload := Unit                                     Put payload types
put-req (k, _)(_, σ) :=                                     At client: Put Request
  ret ⟨∘, σ⟩                                                No payload
put-guard (_, _)(_)(_, _, _) :=                             At replica: Put Guard
  ret true
put (k, v)(_)(_, _, ς) :=                                   At replica: Put
  ret ς[k ↦ v]
```

Figure 15: Relaxed Specification

I.1 Read-Your-Writes (RYW)

A client's read sees its own most recent write. The session tracks every put the client has issued; the get-guard requires the responding replica to have applied every put in that set (Fig. 16). The reference implementation stores per-replica state as a function from key to (value, client, timestamp) (Fig. 17).

$\mathbb{I}^*_{RYW}$ (READ-YOUR-WRITES SPECIFICATION)

```
T := Nat                                                    Timestamp
CState :=                                                   Client State
  Set[T × K]                                                Session puts. (T uniquely identifies puts in a client.)
c-init(c) :=                                                Client Initial State
  ∅                                                         No initial puts
RState :=                                                   Replica State
  K ↦ (V × C × T ×                                          Map from each key k to its value v, its origin ⟨c, t⟩,
       Set[T])                                              and timestamps of puts on k before it.
r-init(r, v_0) :=                                           Replica Initial State
  [k ↦ ⟨v_0, c_0, 0, ∅⟩]

Get-req-payload := Set[T]                                   Get payload types
Get-res-payload := Unit
get-req (k)(_, p) :=                                        At client: Get Request
  ret ⟨p|_k, p⟩                                             Session past puts as request payload, State unchanged
get-guard (k)(p)(c, _, ς) :=                                At replica: Get Guard
  let ⟨_, c', t', p'⟩ ← ς(k)
  ret c = c' ⇒ p ⊆ p' ∪ {t'}                                If stored value is from requesting client, his puts not missed.
get (k)(_)(_, _, ς) :=                                      At replica: Get
  let ⟨v, _, _, _⟩ ← ς(k)
  ret ⟨v, ∘, ς⟩                                             No payload
get-res (_, _)(_)(_, p) :=                                  At client: Get Response
  ret p                                                     State unchanged.

Put-req-payload := T × Set[T]                               Put payload types
put-req (k, _)(self, p) :=                                  At client: Put Request
  let t ← |p| + 1                                           Current put timestamp
  let p' ← p ∪ {⟨t, k⟩}                                     The put is added to session puts.
  ret ⟨⟨t, p|_k⟩, p'⟩                                       Session puts are sent as request payload.
put-guard (_, _)(_)(_, _, _) :=                             At replica: Put Guard
  ret true
put (k, v)(⟨t, p⟩)(c, _, ς) :=                              At replica: Put
  ret ς[k ↦ ⟨v, c, t, p⟩]

p|_k := {t | ⟨t, k⟩ ∈ p}
```

Figure 16: Read-Your-Writes Specification

$\mathbb{I}_{RYW}$ (READ-YOUR-WRITES IMPLEMENTATION)	
$T :=$ Nat	Timestamp
CState $:=$	**Client State**
$K \mapsto T$	Latest put per key ($K \times T$ is unique within a client).
c-init$(c) :=$	**Client Initial State**
$[\overline{k \mapsto 0}]$	No initial puts
RState $:=$	**Replica State**
$K \mapsto (V \times C \times T)$	Map from each key k to its value v, and its origin $\langle c, t \rangle$
r-init$(r, v_0) :=$	**Replica Initial State**
$[\overline{k \mapsto \langle v_0, c_0, 0 \rangle}]$	
Get-req-payload $:= T$	Get payload types
Get-res-payload $:=$ Unit	
get-req $(k)(_, p) :=$	At client: **Get Request**
ret $\langle p(k), p \rangle$	Timestamp of latest put as request payload, State unchanged
get-guard $(k)(t)(c, _, \varsigma) :=$	At replica: **Get Guard**
let $\langle _, c', t' \rangle \leftarrow \varsigma(k)$	
ret $c = c' \Rightarrow t = t'$	If stored value is from requesting client, his puts not missed.
get $(k)(_)(_, _, \varsigma) :=$	At replica: **Get**
let $\langle v, _, _ \rangle \leftarrow \varsigma(k)$	
ret $\langle v, \circ, \varsigma \rangle$	No payload
get-res $(_, _)(_)(_, p) :=$	At client: **Get Response**
ret p	State unchanged.
Put-req-payload $:= T$	Put payload types
put-req $(k, _)(self, p) :=$	At client: **Put Request**
let $t \leftarrow p(k) + 1$	Current put timestamp
let $p' \leftarrow p[k \mapsto t]$	The put timestamp is stored as the latest for the key.
ret $\langle t, p' \rangle$	Latest put timestamp sent as request payload.
put-guard $(_, _)(_)(_, _, _) :=$	At replica: **Put Guard**
ret true	
put $(k, v)(t)(c, _, \varsigma) :=$	At replica: **Put**
ret $\varsigma[k \mapsto \langle v, c, t \rangle]$	

Figure 17: Read-Your-Writes Implementation

I.2 Monotonic Reads (MR)

A client's reads respect the order of writes the client has previously observed. The session keeps the set of writes seen so far; subsequent reads must return values applied no earlier than every write in that set (Fig. 18). The reference implementation stores per-replica state as a function from key to (value, client, timestamp); the session keeps, per key and writing client, the latest timestamp read, and the get-guard requires the stored write to be no older than the session's timestamp for its writing client (Fig. 19).

$\mathbb{I}^*_{MR}$ (MONOTONIC READS SPECIFICATION)

$T \coloneqq \mathsf{Nat}$ — Timestamp

$\mathsf{CState} \coloneqq$ — **Client State**

  $\mathsf{Set}[T \times K] \times$ — Session puts. (T uniquely identifies puts in a client.)

  $\mathsf{Set}[C \times T \times K]$ — Session dependencies.

$\mathsf{c\text{-}init}(c) \coloneqq$ — **Client Initial State**

  $\langle \emptyset, \emptyset \rangle$ — No initial dependencies

$\mathsf{RState} \coloneqq$ — **Replica State**

  $K \mapsto (V \times C \times T \times$ — Map from each key K to its value V, its origin $C \times T$

  $\quad \mathsf{Set}[T \times K])$ — and puts before it

$\mathsf{r\text{-}init}(r, v_0) \coloneqq$ — **Replica Initial State**

  $[\overline{k \mapsto \langle v_0, c_0, 0, \emptyset \rangle}]$

$\mathsf{Get\text{-}req\text{-}payload} \coloneqq \mathsf{Set}[C \times T]$ — Get payload types

$\mathsf{Get\text{-}res\text{-}payload} \coloneqq C \times T$

$\mathsf{get\text{-}req}\ (k)(_, \langle p, d \rangle) \coloneqq$ — At client: **Get Request**

  $\mathsf{ret}\ \langle d|_k, \langle p, d \rangle \rangle$ — Session deps for k sent as payload, State unchanged

$\mathsf{get\text{-}guard}\ (k)(d)(c, _, \varsigma) \coloneqq$ — At replica: **Get Guard**

  $\mathsf{let}\ \langle _, c', t', d' \rangle \leftarrow \varsigma(k)$

  $\mathsf{ret}\ d|_{c'} \subseteq d'|_k \cup \{t'\}$ — Stored put from c' doesn't miss observed puts from c'.

$\mathsf{get}\ (k)(_)(_, _, \varsigma) \coloneqq$

  $\mathsf{let}\ \langle v, c, t, d \rangle \leftarrow \varsigma(k)$

  $\mathsf{ret}\ \langle v, \langle c, t \rangle, \varsigma \rangle$ — The stored put returned as a dependency.

$\mathsf{get\text{-}res}\ (k, _)(\langle c, t \rangle)(_, \langle p, d \rangle) \coloneqq$ — At client: **Get Response**

  $\mathsf{ret}\ \langle p, d \cup \{\langle c, t, k \rangle\} \rangle$ — Returned dependency added to session dependencies.

$\mathsf{Put\text{-}req\text{-}payload} \coloneqq T \times \mathsf{Set}[T \times K]$ — Put payload types

$\mathsf{put\text{-}req}\ (k, _)(_, \langle p, d \rangle) \coloneqq$ — At client: **Put Request**

  $\mathsf{let}\ t \leftarrow |p| + 1$ — Current put timestamp

  $\mathsf{let}\ p' \leftarrow p \cup \{\langle t, k \rangle\}$ — The put is added to session puts.

  $\mathsf{ret}\ \langle \langle t, p \rangle, \langle p', d \rangle \rangle$ — Session puts are sent as request payload.

$\mathsf{put\text{-}guard}\ (_, _)(_)(_, _, _) \coloneqq$ — At replica: **Put Guard**

  $\mathsf{ret}\ \mathsf{true}$

$\mathsf{put}\ (k, v)(\langle t, p \rangle)(c, _, \varsigma) \coloneqq$ — At replica: **Put**

  $\mathsf{ret}\ \varsigma[k \mapsto \langle v, c, t, p \rangle]$

$d|_k \coloneqq \{\langle c, t \rangle \mid \langle c, t, k \rangle \in d\}$

$d|_k \coloneqq \{t \mid \langle t, k \rangle \in d\}$

$d|_c \coloneqq \{t \mid \langle c, t \rangle \in d\}$

Figure 18: Monotonic Reads Specification

$\mathbb{I}_{MR}$ (MONOTONIC READS IMPLEMENTATION)

```
T := Nat                                                    Timestamp
CState :=                                                   Client State
  T ×                                                       Session latest put timestamp
  K ↦ C ↦ T                                                 Largest timestamp read for the key from the client.
c-init(c) :=                                                Client Initial State
  ⟨0, ∅⟩                                                    No initial puts and dependencies
RState :=                                                   Replica State
  K ↦ (V × C × T)                                           Map from each key K to its value V, its origin C × T
r-init(r, v_0) :=                                           Replica Initial State
  [k ↦ ⟨v_0, c_0, 0⟩]

Get-req-payload := Set[C × T]                               Get payload types
Get-res-payload := C × T
get-req (k)(_, ⟨t, d⟩) :=                                   At client: Get Request
  ret ⟨d(k), ⟨t, d⟩⟩                                        Session deps for k sent as payload, State unchanged
get-guard (k)(d)(c, _, ς) :=                                At replica: Get Guard
  let ⟨_, c', t'⟩ ← ς(k)
  ret d(c') ≤ t'                                            Stored put from c' doesn't miss observed puts from c'.
get (k)(_)(_, _, ς) :=
  let ⟨v, c, t⟩ ← ς(k)
  ret ⟨v, ⟨c, t⟩, ς⟩                                        The stored put returned as a dependency.
get-res (k, _)(⟨c, t⟩)(_, ⟨t', d⟩) :=                       At client: Get Response
  ret ⟨t', d[⟨k, c⟩ ↦ t]⟩                                   Returned dependency advances session dependencies.

Put-req-payload := T                                        Put payload types
put-req (k, _)(_, ⟨t, d⟩) :=                                At client: Put Request
  let t' ← t + 1                                            Current put timestamp
  ret ⟨t', ⟨t', d⟩⟩
put-guard (_, _)(_)(_, _, _) :=                             At replica: Put Guard
  ret true
put (k, v)(t)(c, _, ς) :=                                   At replica: Put
  ret ς[k ↦ ⟨v, c, t⟩]
```

Figure 19: Monotonic Reads Implementation

I.3 Monotonic Writes (MW)

A single client's writes are observed in order at every replica that observes them. Each replica records, per key, the set of puts already applied; the put-guard rejects any out-of-order delivery (Fig. 20). The reference implementation instead keeps, per key, the latest applied timestamp from each writing client; its put-guard admits only the next put from that client (Fig. 21).

$\mathbb{I}^*_{MW}$ (MONOTONIC WRITES SPECIFICATION)

```
T := Nat                                                        Timestamp
CState :=                                                       Client State
  Set[T × K]                                                    Session puts. (T uniquely identifies puts in a client.)
c-init(c) :=                                                    Client Initial State
  ∅                                                             No initial dependencies
RState :=                                                       Replica State
  K ↦ (V × Set[C × T])                                          Map from each key K to its value V, and puts already applied.
r-init(r, v_0) :=                                               Replica Initial State
  [k ↦ ⟨v_0, ∅⟩]

Get-req-payload := Get-res-payload := Unit                       Get payload types
get-req (_)(_, p) :=                                            At client: Get Request
  ret ⟨∘, p⟩                                                    No payload, State unchanged
get-guard (_)(_)(_, _, _) :=                                    At replica: Get Guard
  ret true
get (k)(_)(_, _, ς) :=                                          At replica: Get
  let ⟨v, _⟩ ← ς(k)
  ret ⟨v, ∘, ς⟩                                                 No payload
get-res (_, _)(_)(_, p) :=                                      At client: Get Response
  ret p                                                         State unchanged.

Put-req-payload := T × Set[T]                                   Put payload types
put-req (k, _)(_, p) :=                                         At client: Put Request
  let t ← |p| + 1                                               Current put timestamp
  let p' ← p ∪ {⟨t, k⟩}                                         The put is added to session puts.
  ret ⟨⟨t, p|_k⟩, p'⟩                                           Session puts for the key are sent as request payload.
put-guard (k, _)(⟨t, p⟩)(c, _, ς) :=                            At replica: Put Guard
  let ⟨_, p'⟩ ← ς(k)
  ret p ⊆ p'|_c                                                 Previous puts of session already applied.
put (k, v)(⟨t, _⟩)(c, _, ς) :=                                  At replica: Put
  let ⟨_, p'⟩ ← ς(k)
  ret ς[k ↦ ⟨v, p' ∪ {⟨c, t⟩}⟩]

p|_k := {t | ⟨t, k⟩ ∈ p}
p|_c := {t | ⟨c, t⟩ ∈ p}
```

Figure 20: Monotonic Writes Specification

$\mathbb{I}_{MW}$ (MONOTONIC WRITES IMPLEMENTATION)	
$T \coloneqq \mathsf{Nat}$	Timestamp
$\mathsf{CState} \coloneqq$	**Client State**
$\quad K \mapsto T$	Latest timestamp of puts for the key.
$\mathsf{c\text{-}init}(c) \coloneqq$	**Client Initial State**
$\quad \overline{k \mapsto 0}$	No initial puts
$\mathsf{RState} \coloneqq$	**Replica State**
$\quad K \mapsto (V \times$	Map from each key K to its value V, and
$\qquad C \mapsto T)$	latest puts applied from sessions.
$\mathsf{r\text{-}init}(r, v_0) \coloneqq$	**Replica Initial State**
$\quad [\overline{k \mapsto \langle v_0, \overline{c \mapsto 0}\rangle}]$	
$\mathsf{Get\text{-}req\text{-}payload} \coloneqq \mathsf{Get\text{-}res\text{-}payload} \coloneqq \mathsf{Unit}$	Get payload types
$\mathsf{get\text{-}req}\ (_)(_, \sigma) \coloneqq$	At client: **Get Request**
$\quad \mathsf{ret}\ \langle \circ, \sigma \rangle$	No payload, State unchanged
$\mathsf{get\text{-}guard}\ (_)(_)(_, _, _) \coloneqq$	At replica: **Get Guard**
$\quad \mathsf{ret}\ \mathsf{true}$	
$\mathsf{get}\ (k)(_)(_, _, \varsigma) \coloneqq$	At replica: **Get**
$\quad \mathsf{let}\ \langle v, _ \rangle \leftarrow \varsigma(k)$	
$\quad \mathsf{ret}\ \langle v, \circ, \varsigma \rangle$	No payload
$\mathsf{get\text{-}res}\ (_, _)(_)(_, \sigma) \coloneqq$	At client: **Get Response**
$\quad \mathsf{ret}\ \sigma$	State unchanged.
$\mathsf{Put\text{-}req\text{-}payload} \coloneqq T$	Put payload types
$\mathsf{put\text{-}req}\ (k, _)(_, p) \coloneqq$	At client: **Put Request**
$\quad t \leftarrow p(k)$	
$\quad \mathsf{let}\ p' \leftarrow p[k \mapsto t + 1]$	The timestamp of the latest put for key is advanced.
$\quad \mathsf{ret}\ \langle t, p' \rangle$	Latest put timestamp for the key is sent as request payload.
$\mathsf{put\text{-}guard}\ (k, _)(t)(c, _, \varsigma) \coloneqq$	At replica: **Put Guard**
$\quad \mathsf{let}\ \langle _, p' \rangle \leftarrow \varsigma(k)$	
$\quad \mathsf{ret}\ p'(c) = t$	Previous puts of session already applied.
$\mathsf{put}\ (k, v)(t)(c, _, \varsigma) \coloneqq$	At replica: **Put**
$\quad \mathsf{let}\ \langle _, p' \rangle \leftarrow \varsigma(k)$	
$\quad \mathsf{ret}\ \varsigma[k \mapsto \langle v, p'[c \mapsto t + 1]\rangle]$	

Figure 21: Monotonic Writes Implementation

I.4 Read-Your-Writes + Monotonic Writes (RYW+MW)

Composition of Read-Your-Writes and Monotonic Writes. RYW+MW is the conjunction of RYW and MW rather than an independent property; an implementation satisfies it iff it refines both, so correctness is the joint refinement obligation. The reference combines RYW's read-side check with MW's compare-and-swap on every put (Fig. 22).

$\mathbb{I}_{RYW\text{-}MW}$ (READ-YOUR-WRITES AND MONOTONIC WRITES IMPLEMENTATION)	
$T :=$ Nat	Timestamp
CState :=	**Client State**
$K \mapsto T$	Latest timestamp of puts for the key.
c-init(c) :=	**Client Initial State**
$\overline{k \mapsto 0}$	No initial puts
RState :=	**Replica State**
$K \mapsto (V \times C \times C \mapsto T)$	Map from each key K to its value V, and its originating sessions C, latest puts applied from sessions
r-init(r, v_0) :=	**Replica Initial State**
$[\overline{k \mapsto \langle v_0, c_0, \overline{c \mapsto 0}\rangle}]$	
Get-req-payload $:= T$	Get payload types
Get-res-payload := Unit	
get-req $(k)(_, p)$:=	At client: **Get Request**
ret $\langle p(k), p\rangle$	Timestamp of latest put as request payload, State unchanged
get-guard $(k)(t)(c, _, \varsigma)$:=	At replica: **Get Guard**
let $\langle _, c', p'\rangle \leftarrow \varsigma(k)$	
ret $c = c' \Rightarrow p'(c') = t$	If stored value is from requesting client, his puts not missed.
get $(k)(_)(_, _, \varsigma)$:=	At replica: **Get**
let $\langle v, _, _\rangle \leftarrow \varsigma(k)$	
ret $\langle v, \circ, \varsigma\rangle$	No payload
get-res $(_, _)(_)(_, p)$:=	At client: **Get Response**
ret p	State unchanged.
Put-req-payload $:= T$	Put payload types
put-req $(k, _)(_, p)$:=	At client: **Put Request**
$t \leftarrow p(k)$	
let $p' \leftarrow p[k \mapsto t + 1]$	The timestamp of the latest put for key is advanced.
ret $\langle t, p'\rangle$	Latest put timestamp for the key is sent as request payload.
put-guard $(k, _)(t)(c, _, \varsigma)$:=	At replica: **Put Guard**
let $\langle _, _, p'\rangle \leftarrow \varsigma(k)$	
ret $p'(c) = t$	Previous puts of session already applied.
put $(k, v)(t)(c, _, \varsigma)$:=	At replica: **Put**
let $\langle _, _, p'\rangle \leftarrow \varsigma(k)$	
ret $\varsigma[k \mapsto \langle v, c, p'[c \mapsto t + 1]\rangle)]$	

Figure 22: Read-Your-Writes and Monotonic Writes Implementation

I.5 Causal Consistency (CC) and Labeled Causal Consistency (LCC)

Causal Consistency (CC). Cross-client causal consistency: every read returns a value whose causal predecessors have all been observed at the responding replica. Dependencies are tracked as explicit sets in the spec state (Fig. 23), which the implementation must bridge to its concrete representation (typically a per-client vector clock plus a compare-and-swap on the writer's own slot during put). Two reference implementations are given: a per-client vector-clock store (Fig. 25) and a Lloyd–Freedman-style protocol (Fig. 26).

Labeled Causal Consistency (LCC). Causal consistency parameterized by per-message topic labels and per-client interest masks. Every put is tagged with a label, and every client declares which labels it cares about; the get-guard requires the responding replica to have observed every dependency whose label is in the requesting client's mask (Fig. 24). To our knowledge this is the first Rocq formalization of LCC and the first verified implementation.

$\mathbb{I}^*_{CC}$ (CAUSAL CONSISTENCY SPECIFICATION)

```
T := Nat                                                          Timestamp
CState :=                                                         Client State
  ⟨Set[C × T × K],                                                Session dependencies from puts.
   Set[Set[C × T × K]]⟩                                           Session dependencies from gets
                                                                  (C × T uniquely identifies puts.)
c-init(c) :=                                                      Client Initial State
  ⟨∅, ∅⟩                                                          No initial dependencies
RState :=                                                         Replica State
  K ↦ (V × C × T ×                                                Map from each key K to its value V, its origin C × T,
       Set[C × T × K])                                            and set of dependencies
r-init(r, v_0) :=                                                 Replica Initial State
  [k ↦ ⟨v_0, c_0, 0, ∅⟩]                                          Empty initial dependencies

Get-req-payload := ⟨Set[C × T × K],                               Get payload types
                    Set[Set[C × T × K]]⟩
Get-res-payload := Set[C × T × K]
get-req (k)(_, ⟨p, G⟩) :=                                         At client: Get Request
  ret ⟨⟨p|_k, G|_k⟩, ⟨p, G⟩⟩                                      Session deps sent as request payload, State unchanged
get-guard (k)(⟨p, G⟩)(c, _, ς) :=                                 At replica: Get Guard
  let ⟨_, c', t', d'⟩ ← ς(k)
  ret forall (λd. ¬(d'|_k ∪ {⟨c', t', k⟩} ⊂ d)) G ∧                Stored value isn't behind any previously read value.
      c = c' ⇒ p ⊆ d'|_k ∪ {⟨c', t', k⟩}                           Read your writes.
get (k)(_)(_, _, ς) :=                                            At replica: Get
  let ⟨v, c, t, d⟩ ← ς(k)
  let d' ← if (c ≠ c_0) d ∪ {⟨c, t, k⟩} else ∅                     The dependencies returned to the session
  ret ⟨v, d', ς⟩                                                  are the entry and its dependencies
get-res (_, _)(d')(_, ⟨p, G⟩) :=                                  At client: Get Response
  ret ⟨p, G ∪ {d'}⟩                                               Received dependencies added to session dependencies.

Put-req-payload := T × Set[C × T × K]                             Put payload types
put-req (k, _)(self, ⟨p, G⟩) :=                                   At client: Put Request
  let t ← |p| + 1                                                 Current put timestamp
  let p' ← p ∪ {⟨self, t, k⟩}                                     The put is added to session dependencies.
  ret ⟨⟨t, p ∪ (⋃ G)⟩, ⟨p', G⟩⟩                                   Session dependencies are sent as request payload.
put-guard (k, _)(⟨t, _⟩)(c, _, ς) :=                              At replica: Put Guard
  let ⟨_, _, _, d'⟩ ← ς(k)
  ret ⟨c, t, k⟩ ∉ d'                                              The new put is not causally before the current stored put.
put (k, v)(⟨t, d⟩)(c, _, ς) :=                                    At replica: Put
  ret ς[k ↦ ⟨v, c, t, d⟩]

d|_k := {⟨c, t, k⟩ | ⟨c, t, k'⟩ ∈ d ∧ k = k'}
D|_k := {d|_k | d ∈ D}
```

Figure 23: Causal Consistency Specification

$\mathbb{I}^*_{LCC}$ (LABELED CAUSAL CONSISTENCY SPECIFICATION)

L — Label type

$label : V \to L$ — Maps each value to its label

$client\text{-}labels : C \to \mathsf{Set}[L]$ — Maps each client to its set of relevant labels

$T \coloneqq \mathsf{Nat}$ — Timestamp

$\mathsf{CState} \coloneqq$ — **Client State**

  $\langle \mathsf{Set}[C \times T \times K \times L],$ — Session dependencies from puts (with labels)

  $\mathsf{Set}[\mathsf{Set}[C \times T \times K \times L]]\rangle$ — Session dependencies from gets (with labels)

$\mathsf{c\text{-}init}(c) \coloneqq$ — **Client Initial State**

  $\langle \emptyset, \emptyset \rangle$ — No initial dependencies

$\mathsf{RState} \coloneqq$ — **Replica State**

  $K \mapsto (V \times C \times T \times$ — Map from each key K to its value V, its origin $C \times T$,

    $\mathsf{Set}[C \times T \times K \times L])$ — and set of label-annotated dependencies

$\mathsf{r\text{-}init}(r, v_0) \coloneqq$ — **Replica Initial State**

  $[k \mapsto \langle v_0, c_0, 0, \emptyset \rangle]$ — Empty initial dependencies

$\mathsf{Get\text{-}req\text{-}payload} \coloneqq \langle \mathsf{Set}[C \times T \times K \times L],$ — Get payload types

  $\mathsf{Set}[\mathsf{Set}[C \times T \times K \times L]]\rangle$

$\mathsf{Get\text{-}res\text{-}payload} \coloneqq \mathsf{Set}[C \times T \times K \times L]$

$\mathsf{get\text{-}req}\ (k)(self, \langle p, G \rangle) \coloneqq$ — At client: **Get Request**

  $\mathsf{let}\ L \leftarrow client\text{-}labels(self)$

  $\mathsf{ret}\ \langle \langle p|_k|_L, G|_k|_L \rangle, \langle p, G \rangle \rangle$ — Label-filtered deps sent as payload; state unchanged.

$\mathsf{get\text{-}guard}\ (k)(\langle p, G \rangle)(c, _, \varsigma) \coloneqq$ — At replica: **Get Guard**

  $\mathsf{let}\ \langle v', c', t', d' \rangle \leftarrow \varsigma(k)$

  $\mathsf{let}\ L \leftarrow client\text{-}labels(c)$

  $\mathsf{ret}\ \mathsf{forall}\ (\lambda d.$ — Stored value not behind any previously

    $\neg((d'|_k \cup \{\langle c', t', k, label(v') \rangle\})|_L \subset d))\ G$ — read value, for client's labels.

  $\wedge\ c = c' \Rightarrow p \subseteq d'|_k \cup \{\langle c', t', k, label(v') \rangle\}$ — Read your writes.

$\mathsf{get}\ (k)(_)(_, _, \varsigma) \coloneqq$ — At replica: **Get**

  $\mathsf{let}\ \langle v, c, t, d \rangle \leftarrow \varsigma(k)$

  $\mathsf{let}\ d' \leftarrow \mathsf{if}\ (c \neq c_0)\ d \cup \{\langle c, t, k, label(v) \rangle\}\ \mathsf{else}\ \emptyset$ — Label of stored value included in returned dependency.

  $\mathsf{ret}\ \langle v, d', \varsigma \rangle$

$\mathsf{get\text{-}res}\ (_, _)(d')(_, \langle p, G \rangle) \coloneqq$ — At client: **Get Response**

  $\mathsf{ret}\ \langle p, G \cup \{d'\} \rangle$ — Received dependencies added to session.

$\mathsf{Put\text{-}req\text{-}payload} \coloneqq T \times \mathsf{Set}[C \times T \times K \times L]$ — Put payload types

$\mathsf{put\text{-}req}\ (k, v)(self, \langle p, G \rangle) \coloneqq$ — At client: **Put Request**

  $\mathsf{let}\ t \leftarrow |p| + 1$ — Current put timestamp

  $\mathsf{let}\ p' \leftarrow p \cup \{\langle self, t, k, label(v) \rangle\}$ — Put with its label added to session dependencies.

  $\mathsf{ret}\ \langle \langle t, p \cup (\bigcup G) \rangle, \langle p', G \rangle \rangle$ — All session dependencies sent as request payload.

$\mathsf{put\text{-}guard}\ (k, v)(\langle t, _ \rangle)(c, _, \varsigma) \coloneqq$ — At replica: **Put Guard**

  $\mathsf{let}\ \langle _, _, _, d' \rangle \leftarrow \varsigma(k)$

  $\mathsf{ret}\ \langle c, t, k, label(v) \rangle \notin d'$ — The new put has not already been applied.

$\mathsf{put}\ (k, v)(\langle t, d \rangle)(c, _, \varsigma) \coloneqq$ — At replica: **Put**

  $\mathsf{ret}\ \varsigma[k \mapsto \langle v, c, t, d \rangle]$

$d|_k \coloneqq \{\langle c, t, k, l \rangle \mid \langle c, t, k', l \rangle \in d \wedge k = k'\}$

$D|_k \coloneqq \{d|_k \mid d \in D\}$

$d|_L \coloneqq \{\langle c, t, k, l \rangle \mid \langle c, t, k, l \rangle \in d \wedge l \in L\}$ — Filter dependencies to the label set L

$D|_L \coloneqq \{d|_L \mid d \in D\}$

Figure 24: Labeled Causal Consistency Specification. Causal consistency — but per topic. Every put is tagged with a label (think: Kafka topic, Slack channel, pub/sub subject). Every client declares which labels it cares about. Causal delivery for a client is enforced only for labels the client is interested in.

$\mathbb{I}_{CC_1}$ (CAUSAL CONSISTENCY IMPLEMENTATION 1)

```
T := Nat                                                        Timestamp
CState :=                                                       Client State
  C ↦ T                                                         Session vector-clock
c-init(c) :=                                                    Client Initial State
  [c ↦ 0]                                                       No initial dependencies
RState :=                                                       Replica State
  ⟨K ↦ V,                                                       Store
   C ↦ T⟩                                                       Received vector-clock (one for all keys)
r-init(r, v0) :=                                                Replica Initial State
  ⟨[k ↦ v0], [c ↦ 0]⟩                                           Empty initial dependencies

Get-req-payload := C ↦ T                                        Get payload types
Get-res-payload := C ↦ T
get-req (_)(_, d) :=                                            At client: Get Request
  ret ⟨d, d⟩                                                    Session vector-clock (1) sent as request payload (2) remains unchanged
get-guard (k)(d)(_, _, ς) :=                                    At replica: Get Guard
  let ⟨_, r⟩ ← ς
  ret r ≥ d                                                     Pointwise comparison of vector clocks. Store is ahead of session
get (k)(_)(_, _, ς) :=                                          At replica: Get
  let ⟨s, r⟩ ← ς
  let v ← s(k)
  ret ⟨v, r, ς⟩
get-res (_, _)(r)(_, d) :=                                      At client: Get Response
  ret max(d, r)                                                 Pointwise max of vector-clocks

Put-req-payload := C ↦ T                                        Put payload types
put-req (_, _)(self, d) :=                                      At client: Put Request
  let d' ← d[self ↦ d(self) + 1]                                Session dependencies advanced.
  ret ⟨d, d'⟩                                                   Session vector-clock is sent as request payload.
put-guard (_, _)(d)(c, _, ς) :=                                 At replica: Put Guard
  let ⟨_, r⟩ ← ς
  ret r ≥ d ∧ r(c) = d(c)                                       Store satisfies the dependencies. The new put advances the store.
put (k, v)(d)(c, _, ς) :=                                       At replica: Put
  let ⟨s, r⟩ ← ς
  let r' ← r[c ↦ d(c) + 1]                                      Received advanced
  s' ← s[k ↦ v]
  ret ⟨s', r'⟩
```

Figure 25: Causal Consistency Implementation 1 (Inspired by [2]).

$\mathbb{I}_{CC_2}$ (CAUSAL CONSISTENCY IMPLEMENTATION 2)

```
T := Nat                                                    Timestamp
CState :=                                                   Client State
  T × List[C × T]                                           Put timestamp, and Session dependencies
c-init(c) :=                                                Client Initial State
  ⟨0, ∅⟩                                                    Timestamp zero. No initial dependencies.
RState :=                                                   Replica State
  ⟨K ↦ C × T × V,                                           Store (C × T uniquely identifies puts.)
   C ↦ T⟩                                                   Received vector-clock
r-init(r, v_0) :=                                           Replica Initial State
  ⟨[k ↦ ⟨c_0, 0, v_0⟩], [c ↦ 0]⟩                            Nothing initially received

Get-req-payload := List[C × T]                              Get payload types
Get-res-payload := C × T
get-req (_)(_, σ) :=                                        At client: Get Request
  let ⟨_, d⟩ ← σ
  ret ⟨d, σ⟩                                                Session dependencies sent as request payload
get-guard (_)(d)(_, _, ς) :=                                At replica: Get Guard
  let ⟨_, r⟩ ← ς
  ret forall (λ(c, t). r(c) ≥ t) d                          Store satisfies the dependencies
get (k)(_)(_, _, ς) :=                                      At replica: Get
  let ⟨s, _⟩ ← ς
  let ⟨c, t, v⟩ ← s(k)
  ret ⟨v, ⟨c, t⟩, ς⟩
get-res (_, _)(⟨c, t⟩)(_, σ) :=                             At client: Get Response
  let ⟨t', d⟩ ← σ
  let d' ← d :: ⟨c, t⟩
  ret ⟨t', d'⟩

Put-req-payload := List[C × T]                              Put payload types
put-req (_, _)(c, σ) :=                                     At client: Put Request
  let ⟨t, d⟩ ← σ
  let t' ← t + 1
  let d' ← [⟨c, t'⟩]
  ret ⟨d, ⟨t', d'⟩⟩                                         Session dependencies sent as request payload.
put-guard (_, _)(d)(c, _, ς) :=                             At replica: Put Guard
  let ⟨_, r⟩ ← ς
  ret forall (λ(c, t). r(c) ≥ t) d ∧                        Store satisfies the dependencies
      lookup (c, d, 0) = r(c)                               The new put advances the store
put (k, v)(_)(c, _, ς) :=                                   At replica: Put
  let ⟨s, r⟩ ← ς
  let t ← r(c) + 1                                          Current put timestamp
  let r' ← r[c ↦ t]
  s' ← s[k ↦ ⟨c, t, v⟩]
  ret ⟨s', r'⟩
```

Figure 26: Causal Consistency Implementation 2 (Inspired by [33]).

I.6 Chapar Causal Consistency

For Chapar CC we use Chapar's published causal-consistency specification [31]; we refer the reader to the original paper for its formal definition. Per-operation cost comparison appears in App. D.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The abstract and §1 state the contribution; §5 and the appendices substantiate each claim.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

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

   Answer: [Yes]

   Justification: Discussed in §6 (Limitations paragraph) and App. H.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [N/A]

Justification: The paper does not state theoretical theorems with informal proofs; IDS produces Rocq artifacts whose proofs are machine-checked by the kernel (App. B).

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: §5.1 specifies model, baselines, and the shared per-spec budget policy; App. D details the harness, cluster, and workload parameters; App. B reports per-spec implementation choices and lemma classes.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [Yes]

Justification: The IDS suite specifications are documented in App. A and App. I; IDS system code is available at https://github.com/skydiscover-ai/skydiscover under an open-source license.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (https://neurips.cc/public/guides/CodeSubmissionPolicy) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

Answer: [Yes]

Justification: Specified in §5.1 and App. D.

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
- The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

Answer: [Yes]

Justification: Each cell of Table 1 aggregates three independent runs; relative standard deviation across runs is at most 7% (caption).

Guidelines:

- The answer [N/A] means that the paper does not include experiments.
- The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).

- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: App. D specifies the cluster (5 `e2-standard-2` VMs, Debian 12); per-spec wall-clock and dollar cost are in Table 1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The work involves no human subjects, no scraped data, and no high-misuse-risk model.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: Discussed in App. H.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.

- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: The paper releases verified Rocq implementations and a synthesis system; outputs are machine-checked against formal specifications and pose no meaningful misuse risks.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: Existing assets (Chapar [31], Rocq [9], the verification benchmarks in App. E, and the LLM APIs in §5.1) are properly cited and used per their licenses (e.g., MIT for Chapar, IronFleet, Verus-Bench, VeriSoftBench, and CloverBench; Apache-2.0 for DafnyBench, VERINA, and AlgoVeri; LGPL-2.1 for Rocq).

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.

- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

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

    Answer: [Yes]

    Justification: The IDS suite spec dataset (App. A) is documented in the paper, with the spec listings in App. I; the IDS system is available at `https://github.com/skydiscover-ai/skydiscover` under the Apache-2.0 license.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: The paper does not involve crowdsourcing or human-subjects research.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: The paper does not involve human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [Yes]

    Justification: IDS' core synthesis loop uses LLMs as agents (Codex with GPT-5.4; Claude Code with Claude Opus 4.6); model versions are stated in §5.1 and the multi-agent architecture is described in §4 and App. F.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.