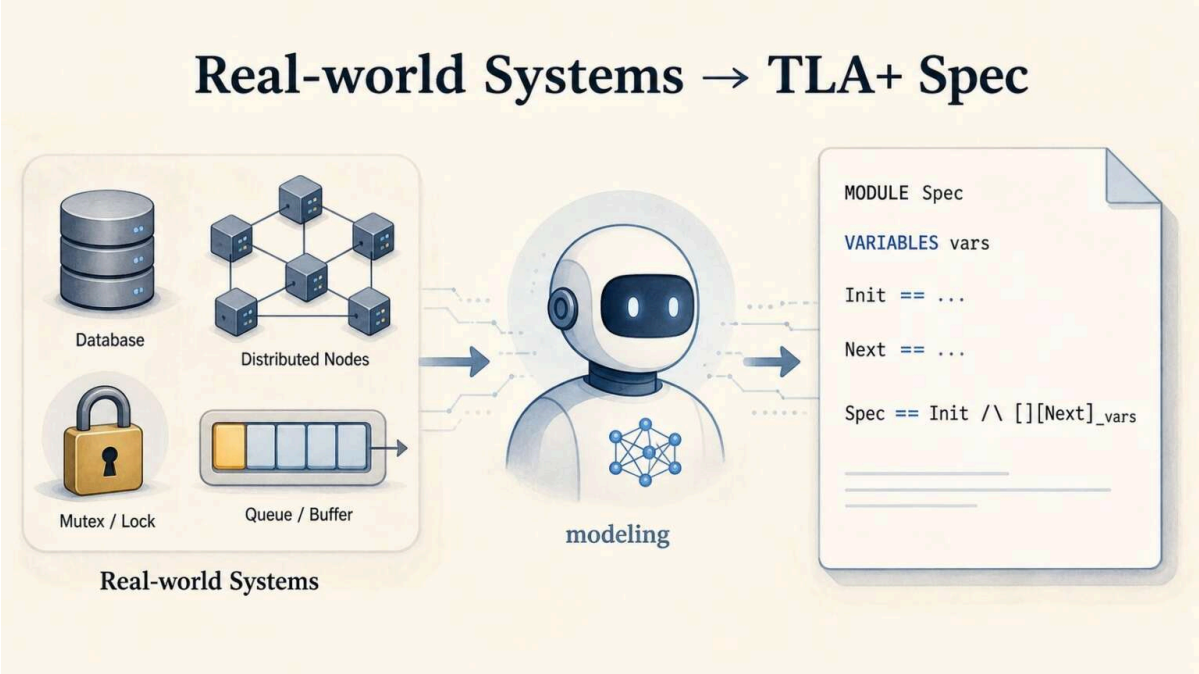

Can LLMs model real-world systems in TLA+?

May 7, 2026 by Qian Cheng, Ruize Tang, Emilie Ma, Finn Hackett, Peiyang He, Yiming Su, Ivan Beschastnikh, Yu Huang, Xiaoxing Ma, and Tianyin Xu

Editors' note: *AI has been actively pushing the frontier of applied formal methods for computing systems. In this article, the Specula team wrote about their experience of evaluating LLMs on modeling system code, the basic capability for agentic model checking, using TLA+, a specification language for concurrent and distributed systems. The article is the 7th blog in The Next Horizon of System Intelligence series.*

Several months ago, we asked Claude to write a TLA+ specification (spec) for Etcd's Raft implementation. It passed syntax checks, ran through the TLC model checker, and at first glance looked like a polished formal model. Then we noticed something: this looks like the spec from the Raft paper, and it has very little to do with Etcd-specific details. What Claude produced was not a spec for Etcd. It was a spec from the appendix of the Raft paper. Later, we kept coming back to the question: how do we tell whether the AI is faithfully modeling a computing system, or just reciting the system's reference paper?

This question gets harder to answer as Large Language Models (LLMs) keep improving. LLMs have seen nearly every TLA+ example online. Asking one to "write a Raft spec" is almost the same as asking it to recall something from training. Asking it to "write Etcd's spec", a spec that reflects how Etcd actually decomposes its atomic actions and evolves its state, is a different problem entirely. It tests whether the LLMs can abstract logic out of a complex implementation and turn that abstraction into a correct formal model.

SysMoBench is our attempt to differentiate the two via an automated benchmark.

What is SysMoBench?

SysMoBench provides eleven systems to an LLM and automatically evaluates the TLA+ specs it generates.

System	Type	Desc.	Source Lang.	Source LoC	TLA$^+$ LoC
Asterinas Spinlock	Concurrent	Synchronization	Rust	213	151
Asterinas Mutex	Concurrent	Synchronization	Rust	186	219
Asterinas Rwmutex	Concurrent	Synchronization	Rust	395	250
Asterinas Ringbuffer	Concurrent	Data Structure	Rust	615	123
Etcd Raft	Distributed	Consensus (Raft)	Go	2,159	385
Redis Raft	Distributed	Consensus (Raft)	C	2,394	349
Xline CURP	Distributed	Replication (CURP)	Rust	4,064	100
ZooKeeper FLE	Distributed	Leader Election	Java	5,360	141
PGo dqueue	Distributed	Distributed Queue	Go	175	75
PGo locksvc	Distributed	Lock Server	Go	281	93
PGo raftkvs	Distributed	Consensus (Raft)	Go	3,163	508

Table 1: Systems used as artifacts in SysMoBench

The eleven systems span concurrent synchronization and distributed protocols. For each task, we provide source code, a trace-collection harness,

and an invariant template.

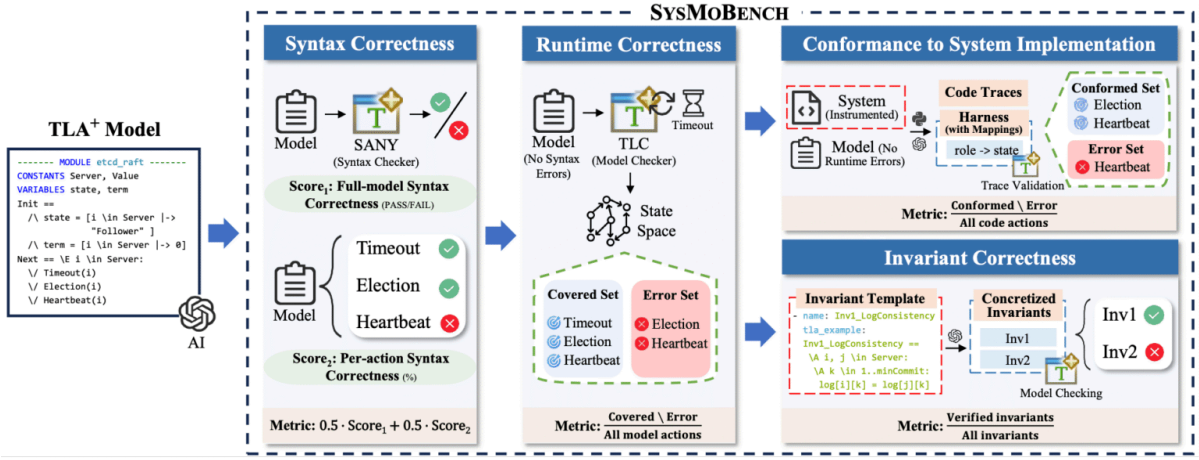

Figure 1: Overview of SysMoBench

Evaluation runs in four phases:

1. The syntax phase checks whether a spec compiles.
2. The runtime phase checks whether TLC can execute it without error.
3. The conformance phase uses trace validation, the common method for checking spec-code consistency: execution traces from code are compared against the model.
4. The invariant phase checks whether the spec satisfies key safety and liveness properties.

Together, these four phases expose gaps between a spec that just recites the textbook and one that actually models the system. Each phase produces action- or invariant-granular diagnostics rather than a single aggregate score, so we can see exactly on which action or invariant the spec misaligns with the implementation.

LLM Modeling Patterns

When we run today's leading LLMs on SysMoBench — Claude, GPT, Gemini, DeepSeek, Kimi, Qwen, etc — a recurring pattern emerges. Their specs do quite well in the first two phases (syntax and runtime). Most compile cleanly, and many run through TLC without error. But, once evaluation reaches the conformance phase, two systematic forms of "textbook modeling" become apparent: (1) the spec enters states that a real system can never reach, and (2)

the spec fails to reach states that a real system always reaches. These two failure modes show up directly in the conformance and invariant scores: even the latest leading LLMs average around 46% on conformance and 41% on invariant, compared to near-perfect scores on syntax.

In the cases where the spec enters states the real-world systems never reach, the LLMs write the spec following a common formalization template that does not match the system's actual data structure, and as a result, the spec admits transitions that produce states that real systems would never produce. A concrete example comes from Claude Sonnet's spec for ZooKeeper Fast Leader Election (FLE).

Figure 2: Two failure modes in an LLM-generated ZooKeeper FLE spec

In ZooKeeper code, each server's recvset is a map keyed by the sender: when the same peer sends a new vote in a new round, it overwrites the peer's old vote. Sonnet wrote this as a set union, recvVotes' = recvVotes ∪ {newVote}, keeping both the old and the new vote. The "accumulate all votes as evidence" pattern appears in many formal-methods textbooks, but it does not match ZooKeeper's semantics. As a result, whenever a peer's vote changes across rounds (which happens routinely in ZooKeeper's elections), the spec's post-state contains both the old and the new vote while the real system retains only the new vote. Once a downstream quorum check depends on vote counts, the spec encounters a state real code never enters.

In the case where the spec fails to reach states real systems reach, the LLM merges operations that span multiple steps in the code into a single atomic guard, making transitions impossible in the spec. In the same Sonnet's ZooKeeper spec, the HandleNotification action carries a guard, m.electionEpoch <= logicalClock[s], that checks if the incoming message's epoch is higher than the local logicalClock. If so, the action is disabled. But, ZooKeeper's code doesn't work this way. When a server receives a message with a higher electionEpoch, it first increases its own logicalClock to match, then processes the message. These two steps happen in that order in the code. However, the LLM fused them into a single guard and, in doing so, erased states that code enters in every election round (local epoch=1, incoming epoch=2).

The above two patterns share a common cause. The LLMs produce structurally complete, type-correct TLA+ modules, but these are written in textbook formalization templates rather than reflecting the actual implementation. The LLM knows what Raft and ZAB look like as protocols, but it does not know how Etcd or ZooKeeper splits a particular action across multiple steps. This is exactly why syntax and runtime evaluation criteria are not enough. The specs produced by an LLM all pass the SANY TLA+ parser, as they are structurally complete and syntactically clean. To distinguish "modeling Etcd" from "reciting the Raft paper," evaluation has to reach the conformance and invariant phases, asking each action whether its transition matches the state changes the system actually produces at runtime.

Transition Validation: Reading Specs at Action Granularity

All phases in SysMoBench produce per-action or per-invariant diagnostics beyond a single aggregate score. Instead of compiling the whole module, the syntax phase runs per-action decomposition to localize which action is malformed. The runtime phase analyzes per-action coverage of the state space rather than only whether TLC can execute the spec at all. The invariant phase verifies each invariant separately, with an LLM translating a fixed template into runnable invariants for each generated spec. The conformance phase uses what we call *Transition Validation*, and it most directly reveals the gap between "reciting the textbook protocol" and "modeling the system."

The idea is simple. We collect execution traces from real runs of the system, then cut each trace into a series of “transition windows”. A window is a triple: (pre_state, action, post_state). Each window is fed independently to TLC, which checks whether the spec’s action can move from pre_state to post_state. The output is not a single score but rather a per-action breakdown, with one pass rate per action.

As one concrete example: when we run Transition Validation on Asterinas RwMutex, the output is a per-action scorecard, detailing one pass rate per action. Coarse-grained evaluation cannot provide this kind of diagnostics. Whereas a single score only tells you if the spec passed or failed, Transition Validation tells you not just which action misaligns, but which specific state transition fails, pinned to a particular window in the trace.

Findings: Where the Scores Diverge

Running leading LLMs across the eleven systems shows that LLMs are great at producing correct TLA+ syntax but struggle to ensure conformance and appropriate invariants.

Figure 3: Per-phase scores across 11 LLMs on SysMoBench (sorted by overall); The ranks of different LLMs can be found at the leaderboard of SysMoBench

Most LLMs cluster near 100% in the syntax phase: nearly every frontier LLM can write a syntactically valid TLA+ spec. The runtime phase already shows divergence, ranging from 30% to 92%. The real gap opens up in conformance

and invariance. In the invariant phase, the weakest LLM lands at 16% while Gemini 3.1 reaches 81%.

This pattern of poor conformance and invariant scores is consistent at the per-system level. Figure 4 illustrates this using Claude Sonnet-4.6 as a representative model. On simpler systems (Asterinas Spin, Mutex, RwMutex), nearly every LLM completes phase 1 (syntax) through phase 4 (invariant) with high scores. On complex distributed systems (Etcd, RedisRaft, CURP, PGo raftkvs), LLMs reliably score 100% or near-100% in phase 1 (syntax) but start breaking down from phase 2 (runtime) onward;. Some cannot get TLC to run at all, and those that do land between 10% and 50% for their conformance and invariant scores. Even Claude Sonnet 4.6, one of the most powerful LLMs, scores only 25% overall on RedisRaft and CURP.

Figure 4: Claude Sonnet-4.6's overall score on different systems, sorted by difficulty levels (from easy to hard).

We observe similar behavior across LLMs. Writing a TLA+ module that compiles is achievable, but aligning that module with the actual behavior of a specific system is challenging. The high syntax scores largely reflect the abundance of TLA+ examples in training data. Once evaluation requires decomposing actions the way real code does and matching state to data structures, prior examples stop being useful. Evaluating conformance and invariant granularity separately is what lets us distinguish "writing TLA+" from "modeling a specific system".

Open Challenges

A few problems remain unsolved.

First, window-level evaluation depends heavily on trace sampling. Transition Validation cannot evaluate code paths that traces do not cover. As a concrete example, in one of our runs the AcquireUpReadLock action in Asterinas RwMutex came out to 0 window — not because the spec failed, but because the upread() code path simply hadn't been exercised in that trace. Transition Validation reports this cleanly, but it cannot fill the gap on its own. Systematically expanding trace coverage remains an open problem.

Second, state abstraction inevitably loses information. Representing the variable log as a (logLen, logLastTerm) pair is lossy for actions that inspect log contents in depth, such as HandleAppendEntries checking the term of a particular middle entry. We currently relax these by hand inside Transition Validation modules, without a systematic policy.

Third, generalizability across tasks is a challenge. Adding a new system still requires a hand-written harness, an invariant template, and a Transition Validation module. We are scaffolding automation for this pipeline, but full automation will require more engineering effort.

These are open problems that we hope to tackle (ideally with the community), not problems we have solved. If any of them interests you, we would like to collaborate!

What's Next

We are continuing to iterate on SysMoBench, while building more powerful agentic tools beyond raw LLMs.

In fact, we find that frontier code agents such as Claude Code and Codex already have strong ability in TLA+ modeling of real-world systems: they can autonomously read a target repository, decide what to model, and drive the full specification workflow. We are developing Specula, an agent specialized in TLA+ formal modeling. Specula achieves full conformance and invariant scores on the current SysMoBench tasks (see our leaderboard).

The leaderboard at sysmobench.com is collecting new LLMs and systems on a rolling basis. We welcome contributions of new systems, new LLMs, and new results!

- **Paper:** https://arxiv.org/abs/2509.23130 [1]
- **Code:** https://github.com/specula-org/SysMoBench
- **Leaderboard:** https://sysmobench.com

[1] With the rapid advances of LLMs, the evaluation numbers reported in this article can be different from those in the original paper. In fact, this article is motivated by the recent requests of rerunning SysMoBench with the new batch of models. Our leaderboard keeps track of the new models.

The article is edited by Haoran Qiu and Mike Chieh-Jan Liang.

Artifacts, Blog, Noteworthy

‹ The Long Game: How Agents That Remember Resolve Operational Issues Faster

› Slow Software: The Case for High-latency Systems Development

Recent Posts

Defining AI-Native Systems: Autonomy as Revision Authority

Slow Software: The Case for High-latency Systems Development

Can LLMs model real-world systems in TLA+?

The Long Game: How Agents That Remember Resolve Operational Issues Faster

LDOS: Toward A Learning-Directed Operating System

Archives

August 2026

July 2026

May 2026

March 2026

February 2026

December 2025

November 2025

October 2025

September 2025

August 2025

July 2025

May 2025

February 2025

October 2024

August 2024

July 2024

May 2024

February 2024

January 2024

December 2023

November 2023

August 2023

July 2023

June 2023

March 2023

February 2023

January 2023

November 2022

April 2022

January 2022

December 2021

November 2021

October 2021

September 2021

August 2021

July 2021

April 2021

March 2021

February 2021

January 2021

December 2020

November 2020

October 2020

September 2020

August 2020

July 2020

June 2020

May 2020

April 2020

March 2020

December 2019

November 2019

October 2019

September 2019

August 2019

July 2019

May 2019

March 2019

February 2019

January 2019

November 2018

October 2018

About

This site is maintained by volunteers for ACM SIGOPS. Thank you for visiting! If you have questions about the site, please send a note to our information director.

SIGOPS addresses a broad spectrum of issues associated with operating systems research and development. Members are drawn from a broad community spanning industry, academia, and government. SIGOPS supports many conferences and workshops. Areas of special interest include: interactions with computer architecture,

multiprocessing, distributed, mobile computing, networking, resource management, security, and interprocess communication.