For: technical evaluators

Choosing a Spec Language for Distributed Systems

A technical evaluation brief for teams comparing Wunderspec, TLA+, Quint, FizzBee, and Lean 4

Igor Konnov, Thomas Pani

demo@wunderspec.com

Executive summary. Distributed systems fail through unlucky message orderings, stale observations, retries at the wrong time, partial failures, control-loop behavior, and long state-transition sequences. Specification languages help by making expected behavior explicit. The key choice is not only syntax: it is which artifact your team needs and wants to maintain.

1. Diagnostic: What Artifact Do You Need?

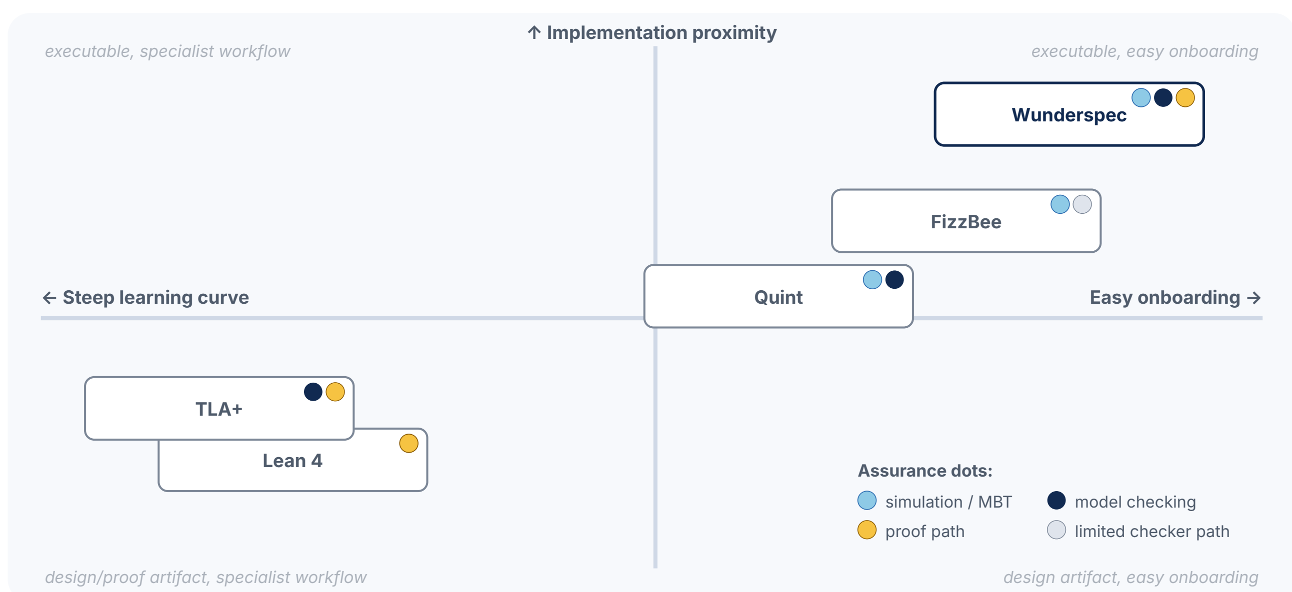
Use this as a quick diagnostic. If your team checks more than one or two boxes, the useful artifact is probably not a Markdown spec alone. You likely need an executable model that can move between design discussion, checking, and implementation testing.

- ☐ We need a model for design/code review
- ☐ We need a shared reference for intended behavior
- ☐ We need a test oracle for expected behavior
- ☐ We need generated tests against the real system
- ☐ We need a simulator for exploring scenarios
- ☐ We need a model-checkable transition system
- ☐ We need proof artifacts for critical invariants
- ☐ We need guardrails for AI-generated code

Diagnostic reading: Wunderspec carries multiple dots because one Python model is intended to support simulation, model-based testing, model checking, and proofs.

2. Spec Language Landscape

This chart compares languages along two workflow dimensions: **learning curve**, from specialist workflow to easier onboarding, and **implementation proximity**, from design/proof artifact to executable CI asset or test oracle. The dots in each box show **assurance modes**: light blue for simulation or MBT, black for model checking (light gray for limited checker support), and gold for proof paths. **Wunderspec carries multiple dots** because one Python model is intended to support simulation, model checking, proof-oriented workflows, and model-based testing.



The chart gives the high-level positioning. The routing table turns that positioning into a first choice; the matrix then shows the practical strengths and trade-offs behind that choice.

Routing guide

Use this table when you already know the main outcome you want: design modeling, modern TLA-style tooling, model-based testing, machine-checked proofs, or a Python-native executable spec.

Tool	Best fit / priority
TLA+	High-level design modeling for concurrent and distributed systems, with the most established ecosystem.
Quint	TLA-style modeling with modern tooling: type checking, REPL, simulator, and symbolic model checking through Apalache.
FizzBee	Accessible formal modeling and model-based testing with Python-like syntax.
Lean 4	Machine-checked proofs, verified mathematics, and verified software as the central goal.
Wunder-spec	Python-native executable specs that connect REPL exploration, simulation, Python and Rust model checking, TLA+ tooling (Apalache, TLC), Lean 4 proofs, and model-based tests against the real system.

Strengths and trade-offs

After the routing table narrows the field, use this matrix to review the trade-offs: what each tool gives you, and what your team may pay in learning curve, extensibility limits, backend constraints, or integration work.

Tool	What it gives you	What to watch
TLA+	Established ecosystem, TLC explicit-state model checking, Apalache symbolic model checking, TLAPS proof path, and strong industrial history.	Unfamiliar syntax and mathematical style for many teams; connecting specs to implementation tests usually requires a lot of plumbing.
Quint	Familiar syntax, type checking, REPL, simulator, symbolic model checking via Apalache, and TLA+ lineage.	Still a distinct specification language; using the model as a test oracle for real services requires additional tooling.
FizzBee	Low learning curve, practical MBT framing, approachable for engineers.	Python-like, but not ordinary Python; formal semantics and model-checking lineage are less explicit than in the TLA+, TLC, and Apalache ecosystem.
Lean 4	Very strong proof assistant, suitable for critical invariants and verified mathematics/software.	High learning curve; not primarily a lightweight modeling and testing workflow.
Wunderspec	Just Python; REPL; Python simulator and checker; Rust multi-core backend; TLA+ export; Apalache path; Lean 4 extraction; MBT focus.	Newer toolchain; evaluate maturity, documentation, and fit on a real subsystem.

Use this as a guide to fit, not as a universal ranking. TLA+ and Lean 4 remain strong choices when the center of gravity is formal design verification or proof. Wunderspec is positioned for teams that want the spec to become an executable engineering asset without giving up paths to stronger backends. This positioning is informed by our prior work on TLA+, Apalache, and Quint.

3. Fit Check

Wunderspec is strongest when the hard part is behavioral correctness, not just input coverage. It is less compelling when the risk is already well covered by ordinary tests or when the organization has committed to a different formal workflow.

Strong fit

- complex state or state machines
- protocols, schedulers, control planes, reconcilers, or distributed workflows
- invariants that must always hold
- failures from ordering, retries, partial failure, crashes, stale reads, or timing
- no executable reference for correct behavior
- need model-based tests against the real implementation

Probably not first

- mostly CRUD behavior
- examples and integration tests already cover the meaningful risk
- the team only wants a written design document
- theorem proving is the main goal from the start
- the organization has already deeply standardized on TLA+ or Lean 4

4. Why Wunderspec?

Wunderspec is a Python-native specification toolkit. Write the model in Python, use it as an executable reference, then choose the backend that fits the job: local simulation and checking, TLA+ tooling such as TLC or Apalache, Lean 4 proof work, or model-based tests against the real system.

1. **It is just Python.** Python is familiar, extensible, operationally boring, and a strong target for AI-assisted development. Teams can read it, debug it, package it, run it in CI, and extend it with domain-specific helpers.
2. **It supports local exploration.** A REPL, Python simulator, and Python model checker keep the modeling loop close to ordinary command-line development: run small scenarios, inspect states, and iterate without turning every question into a batch job.
3. **One model targets multiple backends.** The same source model can support REPL exploration, simulation, Python checking, compiled Rust checking, TLA+ export for Apalache and TLC, Lean 4 extraction, and model-based tests against the implementation.
4. **It connects spec to implementation.** The model can generate happy-path as well as adversarial scenarios and act as an oracle against the real system.
5. **It preserves escape hatches.** Start with the lightweight Python path; move to Rust checking, TLC, Apalache, or Lean 4 when the subsystem justifies stronger tooling. In the worst case, the Python model is still a clear source artifact that humans or AI tools can translate into another formalism.

5. Spec-Driven Development with AI Agents

AI agents make executable specifications more important, not less. A Markdown spec can guide generation, but it cannot check behavior. An executable model gives both humans and agents a concrete reference: generate scenarios, check invariants, run CI, and reject changes that drift from intended behavior.

Executable specs help

- make intended behavior explicit
- generate scenarios the agent did not test
- act as an oracle against implementation changes
- run in CI after AI-generated patches

Python helps

- familiar to engineers
- well represented in model training data
- easy to run in ordinary tooling and CI
- a practical target for agent-assisted authoring

The control question: What executable reference keeps the AI inside the intended behavior?

6. Adoption Path

Evaluate Wunderspec as a Python-native front end with multiple verification and testing paths. A typical adoption path looks like this:

Step	Description
1. Write an executable model in Python	Model state, actions, invariants, and the environment. The first goal is clarity, not proof.
2. Explore with REPL and simulator	Inspect behavior, run scenarios, understand transitions, and refine the model.
3. Check locally	Use the Python model checker for tight feedback while the model evolves.
4. Scale exploration	Compile to the Rust multi-core checker when local checking is not enough.
5. Use TLA+ tooling	Transpile to TLA+ for TLC or Apalache. TLC gives explicit-state checking; Apalache gives symbolic approaches.
6. Extract proof obligations	Move critical invariants into Lean 4 when proof is worth the cost.
7. Test the implementation	Use the model as a scenario generator and oracle for model-based tests against the real system.

Want to Try It on a Real Subsystem?

Send a short message. Rough notes are enough:

1. What subsystem are you considering?
2. What behavior must never break?
3. What makes it hard to test today?
4. What invariants or scenarios would you like the model to cover?
5. Would the model need to drive tests against a real implementation?

> **We will reply with a short recommendation:** which workflow to try first, which tools are likely poor fits, and what a small evaluation model should cover.

→ demo@wunderspec.com

About us

Igor Konnov, PhD, and Dr. Thomas Pani work on executable specifications, model checking, model-based testing, and fuzzing for distributed systems. Their work spans TLA+, Apalache, Quint, Lean 4, protocol verification, conformance testing, and practical test generation against real systems.

Appendix: Detailed Comparisons

A.1 Wunderspec vs. TLA+

TLA+ is the reference point for many distributed-system specification efforts. It is powerful, precise, and supported by established tools such as TLC and TLAPS. Wunderspec does not try to erase TLA+: it uses TLA+ transition-system logic, wrapped in a Python library with accessible syntax. The difference is workflow: TLA+ is excellent when the model is primarily a formal design artifact and your team already knows TLA+; Wunderspec is designed when the model should also become a live engineering artifact.

Choose TLA+ directly if

- your team already has TLA+ expertise
- the main goal is high-level design verification
- the model does not need custom domain logic
- implementation testing is secondary
- your team is equipped to actively manage model drift
- the organization already has a TLA+ workflow

Choose Wunderspec if

- your team wants to start in ordinary Python
- the model should run in CI and be easy to extend
- you want local REPL and simulator workflows
- you want model-based tests against the implementation
- you want optional export to TLA+ for TLC or Apalache
- AI-assisted spec work is part of the workflow

A.2 Wunderspec vs. Quint

Quint and Wunderspec both offer a modern surface over TLA+ transition-system logic. Quint is a dedicated specification language with type checking, a REPL, a simulator, and symbolic model checking through Apalache. Quint was previously designed by the Apalache team, including Igor and Thomas; Wunderspec continues that line of work in ordinary Python, so the model can live in Python tooling, packaging, CI, and implementation-test workflows.

Choose Quint if

- you want a clean, modern TLA-style language
- you value static specification-language tooling
- your team likes TLA+ style modeling but wants a dedicated modern syntax
- the work stays mostly within the specification ecosystem

Choose Wunderspec if

- you want the spec to be Python code
- you want the stability of Python's module system, packaging, and tooling
- you need ordinary Python helper code
- you want Python simulation and model checking
- you want a Rust multi-core checker path
- you want TLA+, Apalache, Lean 4, and MBT paths from one Python model

A.3 Wunderspec vs. FizzBee

FizzBee and Wunderspec are close in spirit: both care about accessibility, Python familiarity, and practical modeling for engineers. FizzBee presents itself as Python-like and emphasizes model-based testing. The key distinction is how much formal-methods machinery sits behind the model. Wunderspec uses TLA+ transition-system semantics and has a path to TLC and Apalache. FizzBee emphasizes accessibility and model-based testing, but its formal semantics and model-checking lineage are less explicit. Wunderspec is also not merely Python-like. It is Python.

Choose FizzBee if

- you want a focused, approachable formal modeling environment
- you like Python-like syntax and an integrated MBT story
- you want a contained tool experience
- TLA+ semantics and checker lineage are not central

Choose Wunderspec if

- you want real Python as the modeling language
- you need custom Python extension points
- you want TLA+ style semantics with a path to TLC and Apalache
- one model should survive across simulation, checking, testing, TLA+ export, and proofs
- AI-assisted spec work matters

A.4 Wunderspec vs. Lean 4

Lean 4 is the right tool when proofs are the primary deliverable. It is both a programming language and a proof assistant, designed for formally verified code and mathematics. The trade-off is time to value: modeling and proving in Lean 4 usually requires proof-engineering work and is not a quick iteration loop for distributed-system design exploration. Wunderspec starts with executable models, exploration, simulation, checking, and test generation, but keeps a path to Lean 4 when parts of the model or invariants deserve proof.

Choose Lean 4 directly if

- ✓ your team already knows Lean 4
- ✓ proof is the primary output
- ✓ you are willing to invest in proof engineering
- ✓ you do not need quick simulation/checking iteration
- ✓ the model is closer to mathematics or verified software than a testing workflow

Choose Wunderspec if

- ✓ you want testing and exploration first
- ✓ you want a proof path later
- ✓ the model must stay accessible to non-proof specialists
- ✓ the near-term value is behavioral bug finding and test generation