Case Study: Li Langverse

A high-performance programming language built for AI agents — near-C++ speed, safer code, fewer tokens

Julian M. Kleber | Founder | Architect | gitlab.lilangverse.xyz

julian.kleber@sail.black | Last updated: July 19, 2026

Outcome snapshot

- **Speed**: compiled Li programs are required to stay **within 80% of C++ speed** on the project's published benchmarks (measured ratios, not marketing claims).
- **AI cost**: a denser query language for agents (`liq`) used ∼**36% fewer tokens than handwritten SQL** in an 18-scenario audit (629→405 tokens).
- **Safety**: memory and concurrency mistakes (including data races) are **rejected when you compile**, not found later in production; functions carry explicit pre/post-condition contracts.
- **Toolchain**: package manager and tests share one quality bar (≥**80% test coverage** before a package can be published), plus a public benchmarks dashboard and shared Cursor agent automation.

CONTEXT

Most language ecosystems grow as loosely coupled tools. Specs drift, install steps diverge, and AI coding agents reinvent the same rules in every repository. Li Langverse keeps the language *and* its surrounding tools under one roadmap: compiler, packages, tests, containers, servers, databases, and Cursor agents share the same product rules.

In plain terms: Li is for teams that need **native performance** (think scientific computing, systems software, agent backends) without accepting "move fast and crash later," and for AI agents that should spend fewer tokens getting the same work done.

PROBLEM DEFINITION

Without an org-level specification, teams face:

- compiler and tooling divergence that breaks install and CI flows,
- unsafe concurrency accepted too late (at runtime) instead of rejected when the program is built,
- AI agent / prompt drift across dozens of repositories,
- no common publish quality bar for packages or benchmarks.

APPROACH

Li Langverse treats the language and the org as one product:

- **Correctness first**: programs declare what must be true before and after a function runs; the compiler refuses parallel code that can race on shared memory.
- **Toolchain completeness**: package manager, test runner, containers, HTTP server, and database layers are first-class repos — not afterthoughts.

- **Shared agent-kit**: one roadmap owns Cursor rules, hooks, and skills; other repos sync them instead of forking policy.
- **Measurable quality bars**: tests enforce coverage before packages publish; a benchmarks dashboard shows whether speed regressions pass or fail.

SYSTEM OUTLINE

- **lic** — the Li compiler (written in C++): type checking, memory/borrow safety, and an LLVM backend that produces native machine code.
- **lip** / **lit** — package resolve/publish and the test runner with the coverage publish bar.
- **liq** / **lidb** — database stack plus an agent-oriented query dialect designed to be shorter (and safer) than free-form SQL for LLMs.
- **roadmap** — architecture decisions, milestones, and the shared Cursor agent-kit.
- **benchmarks** — aggregated status dashboard and scripts that check results before releases.
- **li-cursor-agents** — supervisor UI (Next.js + Supabase) for Cursor SDK automation; CI uses a mock backend so pipelines do not spend on live model APIs.
- **Containers / servers / OS experiments** — language-native infra and longer-horizon kernel work under the same org.

IMPLEMENTATION STACK

- **Compiler**: C++ frontend that emits LLVM intermediate representation, then native binaries.
- **CI / hosting**: GitLab-primary pipelines and public install pages.
- **Agent layer**: Cursor SDK, org agent-kit sync, pull-request-only rollout workflows.
- **Dashboard**: Next.js supervisor UI with Supabase; CI uses a mock LLM backend.
- **Governance**: roadmap milestones and ADRs apply across the org.

RESULTS (VERIFIED, MID-2026)

- **20+ actively maintained repos** in the GitLab org.
- ≥**80% line coverage** required before a package can publish to the registry.
- Public benchmarks dashboard at li-langverse.github.io/benchmarks (includes the within-80%-of-C++ speed rule where the plan requires it).
- Cursor agent supervisor ships with a **mock CI backend** so pipelines stay deterministic without LLM spend.
- Measured ∼**36% token savings** for agent queries via `liq` versus handwritten SQL (same audit scenarios).

WHAT THIS CASE STUDY DEMONSTRATES

- Systems architecture for a language *ecosystem*, not just a compiler demo.
- Safety as a product requirement: bad parallel code fails the build.
- Agent economics: denser, safer queries reduce token spend without a separate SQL dialect for humans.
- Org-wide agent automation that stays reproducible in CI.

WHAT I WOULD IMPROVE NEXT

- Tighten end-to-end install stories from compiler builds into package consumers.
- Expand benchmark fixtures so roadmap milestones fail automatically when speed slips.
- Mature kernel/OS experiments behind clearer public milestones.

LINKS

- Org explore: gitlab.lilangverse.xyz/explore
- Compiler: gitlab.lilangverse.xyz/li-langverse/lic
- Benchmarks: li-langverse.github.io/benchmarks