Zero Human Code: Building Operational Software with AI Coding Agents Without Code Inspection

Katsuya Shibuki

2026-01-14

Abstract

AI coding agents can generate code from natural language, but most workflows assume a human who can read and review that code (Vaithilingam, Zhang, and Glassman 2022). What happens when the human cannot? This experience report documents a 45-day project in which I—a PharmD/PhD statistician who cannot read Python code—built an operational MCP server (~190k LOC including tests) using AI agents with **zero human-written code**. I could not inspect the implementation, so code review was impossible. After an initial failure where specification-driven development produced modules that "looked right but did not run," the project pivoted to an **ADR-driven control plane** governing a fast-changing **data plane** (code, tests). The control plane—decision logs (ADRs), policies (rules), procedures (commands), and quality gates—stabilized **inter-module contracts and coordination**. I describe this control plane's minimal components, how failures drove its evolution, and practical patterns for collaborating with AI agents without inspecting code.

Three actionable insights

1. **Contracts before code**: When implementation is opaque and fast-changing, stability comes from explicit inter-module contracts—responsibilities, dependency direction, invariants, and recovery procedures.
2. **Rules for direction; commands for resilience**: Rules constrain probabilistic agent behavior; commands encode repeatable recovery and prevention workflows under inevitable breakdowns.

3. **Control plane separates governance from implementation**: Treat code and tests as the data plane, ADRs/rules/commands/gates as the control plane. Funnel execution through one interface (`make`) and verify via externalized quality gates.

---

1. Introduction

1.1 The collaboration problem

AI coding agents generate impressive amounts of code from natural language prompts. Industry workflows typically assume a human developer who can:

- read the generated code to verify correctness,
- maintain specifications that describe the target system,
- perform code review before merging changes.

But what if the human cannot do any of these? Domain experts—researchers, analysts, clinicians—increasingly want to build software for their own needs without becoming software engineers. The question then becomes: **what prevents a natural-language-first process from collapsing into drift and broken integration?**

In this report, I describe the answer as a separation between a fast-changing **data plane** (implementation artifacts) and a durable **control plane** (decisions, policies, procedures, and gates). As AI agents increasingly outpace human review capacity, this condition is likely to become common rather than exceptional.

1.2 Author context

I am a pharmacist and statistics researcher (PharmD, PhD) specializing in meta-analysis and meta-regression. At the start of this project:

- I had **no Python experience** and still cannot read or trace Python implementation code.
- I do have programming literacy in R, and I can read/write Make and shell scripts for automation.
- Code review was therefore **impossible**—I could not verify whether generated Python code was correct.

The "zero human code" claim is literal: I wrote no production code (`src/`) and no test code (`tests/`). My role was:

1. Request options and trade-offs from the AI agent.
2. Evaluate those options using domain expertise (not code inspection).
3. Make decisions and record them in ADRs.
4. Invoke standardized procedures and verify outcomes via quality gates.

1.3 Research questions

1. Can someone who cannot read code build operational software using AI coding agents?
2. What minimal "control plane" stabilizes such a workflow?
3. What patterns generalize, and what limitations remain?

---

2. Control Plane vs Data Plane

This section introduces the conceptual frame used throughout this report (descriptive, not prescriptive).

2.1 Data plane

The data plane consists of implementation artifacts: source code, tests, configurations, database schemas. These change frequently—often with every agent interaction. In Lyra, the data plane grew to ~190k lines of Python, of which ~105k were tests, and ~160 test files over 45 days. In this report, tests are treated as data-plane artifacts because they change with implementation; the *gate* is the control-plane practice that invokes and interprets them. (This includes generated integration and regression tests.)

2.2 Control plane

The control plane consists of governance artifacts: ADRs, rules, commands, the Makefile interface, and quality gates. These change infrequently and only when decisions change. The control plane is durable context that persists across implementation churn.

---

3. The Lyra Project

3.1 What Lyra does

Lyra ("Lyra: A Local-First MCP Server for AI-Collaborative Desktop Research with Evidence Graph Construction" n.d.) is an MCP (Model Context Protocol) server for AI-assisted research. It provides:

- **Evidence Graph**: Claims linked to source fragments via NLI (natural language inference; supports/refutes/neutral) with full URL traceability.
- **Multi-source search**: Academic APIs (Semantic Scholar, OpenAlex) and web engines.
- **Local-first architecture**: All ML processing runs locally (zero operational cost, full data sovereignty).

The system is stateful (SQLite), asynchronous (job queues), and boundary-heavy (external sites, authentication blocks, browser automation). These properties make **inter-module coordination** and **failure recovery procedures** first-class concerns.

3.2 Scale and artifacts

Although Lyra originated as a personal research tool, it was engineered with operational controls (tests, static analysis, reproducible procedures) and is used in daily desktop research. Over 45 days (initial commit: 2025-11-25; public release: 2026-01-12):

- **~190k lines of code** (Python, including tests), no human-authored code
- **~160 test files**, no human-authored tests
- **17 ADRs** documenting architectural decisions
- **6 rule files** (agent policies)
- **14 command files** (fixed procedures)

Public artifacts with persistent identifiers:

- Software archive (Shibuki 2026): `https://doi.org/10.5281/zenodo.18218531`
- Repository: `https://github.com/k-shibuki/lyra`
- Preprint: `https://doi.org/10.5281/zenodo.18222598`

---

4. Failure: Specification-Driven Development

4.1 Initial approach

The project began with specification documents: a REQUIREMENTS file, section references (§1.2.3), phase markers, and naming conventions. Code comments referenced these sections. This mirrored a familiar product-management practice: define structure, then implement incrementally.

4.2 What broke

The failure mode was not that the agent could not generate modules. The failure was:

1. **Specs became stale**: The agent produced working code, but I could not reliably update specifications to match. Without being able to read the code, I could not verify whether specs matched reality.
2. **Reference systems exploded**: Multiple naming conventions coexisted (sections, phases, letter+digit codes). It became unclear which document was authoritative.
3. **Inter-module coordination collapsed**: Individual modules passed their own tests but failed when combined. I could not debug this by reading code.

The dominant failure mode was not local correctness but **global contract mismatch**. The result: modules that "looked right but did not run."

4.3 Root cause

The underlying problem was a mismatch between:

- **Velocity of change**: How fast the agent modified the codebase.
- **Capacity to maintain specs**: How fast I could verify and update specifications.

When implementation changes faster than specification maintenance, specifications become stale. For someone who cannot read code, specification maintenance capacity is effectively zero for implementation details. Specification-driven development was structurally doomed in this context.

---

5. Pivot: ADR-Driven Control Plane

5.1 The insight

On 2025-12-23, the project pivoted sharply:

- 12 ADRs were created in a single day.
- Legacy specification documents were archived.
- Code references were migrated from spec sections (§) to ADR numbers (ADR-NNNN).

The conceptual change: **ADRs capture why decisions were made, not what the system looks like** (Nygard 2011; Tyree and Akerman 2005). They do not require continuous truth maintenance. New circumstances produce new ADRs; old ADRs remain valid as historical records.

5.2 The control plane components

ADRs alone were insufficient. Stability required four control-plane mechanisms working together to govern contracts:

Component	Implementation	Function
Decision log	`docs/adr/` (17 ADRs)	Durable context; stable reference point
Policy	`.cursor/rules/` (6 files)	Direction for probabilistic agent behavior
Procedure	`.cursor/commands/` (14 files)	Recovery and prevention workflows
Execution UI + Gate	`Makefile` + tests + static analysis	Single interface; externalized verification

In this framing, inter-module contracts are the *governed object*; ADRs, rules, commands, and gates are the mechanisms that specify, enforce, and restore them. Contract boundaries are captured via ADR anchors and `integration-design`.

The Makefile provides a unified execution interface:

```
make test          # Run tests (async, returns RUN_ID)
make test-check    # Check test results (RUN_ID required)
make quality       # Run all quality checks
make up / down     # Start/stop containers
make help          # Show all targets
```

5.3 Why ADRs do not go stale

Aspect	Specification	ADR
Content	Current desired state	Historical decision
Updates	Required continuously	Append-only
Validity	Degrades as system changes	Permanent
For code-illiterate	Must verify implementation matches	Need only understand the decision

ADRs are history, not prophecy. They describe what was decided and why, not what the system currently looks like.

---

6. Governing Probabilistic Agents: Rules vs Commands

In this project, rules and commands were implemented using Cursor's rule/command system; however, the underlying concepts—policy constraints and recovery procedures—are not specific to any single tool.

AI coding agents exhibit probabilistic behavior: the same prompt can produce different outputs. This section describes how I constrained that behavior.

6.1 Rules: direction under probabilistic behavior

Rules increase predictability by reducing degrees of freedom. They constrain what the agent should and should not do.

Rule file	Key constraints
`quality-check.mdc`	No workarounds (`# noqa`, `# type: ignore`); fix errors, don't silence them
`integration-design.mdc`	Design data flow before implementation; verify module interfaces
`debug.mdc`	Timeout required for async operations; state-based diagnosis
`test-strategy.mdc`	Test perspectives table; coverage requirements

Rules are **direction**: they tell the agent which behaviors are acceptable.

6.2 Commands: resilience under inevitable breakdowns

Commands encode repeatable workflows for repair and prevention. They assume breakdowns will occur and provide playbooks.

Command file	Workflow
`/debug`	State-based debugging: reproduce → isolate → diagnose → fix → verify
`/integration-design`	Pre-implementation verification: sequence diagrams, data flow, interface contracts
`/quality-check`	Full quality sequence: lint → typecheck → test → schema validation
`/regression-test`	Targeted verification after changes

Commands are **resilience**: they restore state when things go wrong.

6.3 Why procedures beat conventions

Conventions assume compliance. Procedures assume failure and provide recovery steps.

When I cannot verify compliance by reading code, conventions are unenforceable. Procedures—with explicit entry points, verification steps, and exit criteria—are the only reliable mechanism. The `/debug` command, for example, includes mandatory timeout specifications and state capture steps precisely because ad hoc debugging repeatedly failed. With probabilistic agents, variance is not an exception but a recurring property; procedures provide repeatable recovery paths.

---

7. Contracts as the Human Focus

When I cannot read code, what can I review? The answer: **contracts**.

7.1 The four contract elements

Inspired by Design by Contract (Meyer 1997), I focused on four elements that define module boundaries:

1. **Responsibilities**: What each module does (not how).
2. **Dependency direction**: Who calls whom; what data flows where.
3. **Invariants**: What must always be true (e.g., "every claim links to a source URL").
4. **Recovery procedures**: What to do when a contract is violated.

7.2 How the control plane enforced contracts

- **ADR anchors**: Decisions that define module boundaries (e.g., ADR-0002 defines the three-layer collaboration model; ADR-0005 defines the evidence graph structure).
- **integration-design rule/command**: Pre-implementation verification of interfaces and data flow.
- **Quality gates**: Externalized correctness oracle (tests verify contracts; static analysis verifies type contracts).

7.3 "Review" without code review

I reviewed contracts and intent, not implementation. When a test failed, I asked the agent: "What is this component intended to do?" and compared that intent to the relevant ADR. If they mismatched, I corrected either the ADR (decision was wrong) or the implementation (decision was right but not followed).

This is slower than direct code inspection, but it is **tractable** for someone who cannot read code.

---

8. How Failures Drove Evolution

The control plane did not appear fully formed. Each component was added or strengthened in response to a specific failure mode. Retrospectively, each failure manifested as a contract violation (or inability to determine which contract was in force), and each countermeasure strengthened contract specification, enforcement, or recovery. Each representative commit was selected as the earliest main-branch changeset that introduced the corresponding countermeasure or a major structural revision, verified by git history.

Failure mode	Symptom	Control plane response	Contract element	Representative commits (examples)
Spec staleness	Outdated references; plausible but non-working integration	ADR introduction; reference migration; archive legacy specs	Responsibilities	`651523a`, `89bcebd`, `3b15391`
Coordination collapse	Modules pass alone, fail together	`integration-design` rule/command	Dependency direction	`a554cb5`
Debugging chaos	Hard reproduction; inconsistent procedures	`debug` rule/command: timeouts, state-based diagnosis	Recovery procedures	`16322a0`, `aa61506`, `3659925`
Execution ambiguity	Inconsistent command choice/order	Makefile as unified UI; RUN_ID required	Recovery procedures	`cc9a655`
Gate avoidance	`# noqa`, `# type: ignore` to silence errors	"No workaround" policy; fixed quality commands	Invariants	`d8757f1`, `f8a7f8b`

The control plane grew by **repairing concrete breakdowns**, not by anticipating them. All artifacts—pivot dates, ADRs, rules, commands, Makefile—are traceable via commit history. Descriptively, median daily code churn remained approximately 6k LOC before and after the pivot (no causal claim made). Quantitative evaluation across projects is future work.

---

9. Limitations

9.1 Reading code reduces iteration time

Zero-code authorship is possible but not optimal. When the developer can read code, debugging is faster and mistakes are caught earlier. The control plane compensates for code illiteracy but does not eliminate its cost. This approach substitutes code literacy with contract literacy—the ability to reason about responsibilities, dependencies, and invariants at the interface level. Contract literacy is itself a skill; it is not "no skill required."

9.2 Partial applicability across agents

Rule/command enforcement was implemented in Cursor. Other environments offer analogous mechanisms (e.g., agent-specific tools/skills, custom system prompts, IDE-specific workflows), but nomenclature and enforcement vary. The limitation is not conceptual but infrastructural: no industry standard exists for portable agent governance.

---

10. Implications

10.1 For practitioners

If you want natural-language-first development to be stable:

1. **Define contracts explicitly**: responsibilities, dependency direction, invariants, recovery procedures.
2. **Separate rules from commands**: rules constrain; commands recover.
3. **Funnel execution** through one interface (Makefile) and enforce quality gates.
4. **Invest early** in integration-design and debugging workflows.

10.2 For tool builders

The most leverage is not in better prompting, but in better governance primitives:

- Standardized rule/command enforcement across agents.
- Tighter links between ADRs and verification artifacts.
- Better observability for stateful, async systems.

Governance primitives belong to the control plane; interoperability across agents is the missing infrastructure. Until governance primitives are standardized, control-plane designs remain tool-specific in implementation, even when conceptually portable.

---

11. Conclusion

This experience report shows that "zero human code" development is possible for someone who cannot read the generated code, but it requires a methodology that replaces code review with a development control plane.

In Lyra, ADR-driven development—combined with explicit agent governance (rules for direction, commands for resilience) and externalized verification (quality gates)—stabilized a natural-language-first workflow after an initial specification-driven failure. The human focus shifted from implementation details to **inter-module contracts**: responsibilities, dependencies, invariants, and recovery procedures.

The practical lesson is not that AI can write code. The lesson is that humans can build reliable systems by **controlling how AI writes code**—through decisions, policies, procedures, and gates—without ever reading the code itself.

Data Availability

- **Software archive (Zenodo DOI)**: `https://doi.org/10.5281/zenodo.18218531`
- **Repository (GitHub)**: `https://github.com/k-shibuki/lyra`

The repository includes all control-plane artifacts described in this report: ADRs (`docs/adr/`), agent rules (`.cursor/rules/`), command procedures (`.cursor/commands/`), and the Makefile interface. Readers can inspect the governance mechanisms in full.

References

"Lyra: A Local-First MCP Server for AI-Collaborative Desktop Research with Evidence Graph Construction." n.d. Accessed January 14, 2026. https://doi.org/10.5281/zenodo.18222598.

Meyer, Bertrand. 1997. *Object-Oriented Software Construction*. Second Edition. Prentice Hall Professional Technical Reference.

Nygard, Michael. 2011. "Documenting Architecture Decisions." November 15, 2011. https://www.cognitect.com/blog/2011/11/15/documenting-architecture-decisions.

Shibuki, Katsuya. 2026. "Lyra: A Local-First MCP Toolkit for AI-Collaborative Desktop Research." Zenodo. https://doi.org/10.5281/zenodo.18218531.

Tyree, J., and A. Akerman. 2005. "Architecture Decisions: Demystifying Architecture." *IEEE Software* 22 (2): 19–27. https://doi.org/10.1109/MS.2005.27.

Vaithilingam, Priyan, Tianyi Zhang, and Elena L. Glassman. 2022. "Expectation Vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models." In *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, 1–7. CHI EA '22. New York, NY, USA: Association for Computing Machinery. https://doi.org/10.1145/3491101.3519665.