SISO: A Pure Functional Framework for Rapid AI-Assisted Software Development

Jonathan Bailey
December 2025

Abstract

We present SISO (Stream In, Stream Out), a pure functional event-driven framework that enables large language models to generate production-quality software with minimal human intervention. SISO enforces architectural constraints—immutability, pure functions, and single-responsibility transformations—that align with LLM pattern recognition capabilities. The framework introduces a layered architecture: CORE primitives for event dispatch, a Protocol layer that separates state declaration from state access, and a Resolution layer that quarantines all impurity. We demonstrate the framework through two case studies: Jaa, a complete SQL database engine implemented in both JavaScript and PHP (1,123 passing tests across both languages, ~11,600 LOC), and a Universal Math Gate covering 11 mathematical domains with 216 pattern-rewrite rules in a single gate (~600 LOC). Empirical results show 10–50x development speedup, with the polyglot database implementation validating that the architecture is language-agnostic. Results suggest that appropriate architectural constraints can bridge the gap between AI code generation and production software quality.

1 Introduction

Large language models (LLMs) have demonstrated significant capabilities in code generation, yet struggle to produce complete, correct implementations for non-trivial systems. Common failure modes include incomplete edge case handling, architectural inconsistencies, and subtle correctness bugs that emerge only in integration.

We hypothesize that these failures stem not from fundamental LLM limitations, but from insufficient structural constraints in traditional software architectures. Specifically, imperative programming with mutable state creates exponentially large state spaces that resist systematic enumeration—precisely the domain where LLMs struggle most.

This paper introduces SISO, a framework that inverts the relationship: rather than asking LLMs to navigate arbitrary architectural decisions, we constrain the architecture such that correct implementations become the path of least resistance. The core insight is that pure functional event-driven architectures provide sufficient structure for LLMs to generate complete, correct implementations while maintaining generality across domains.

1.1 Contributions

- A layered pure functional framework (Event/Gate/Stream + Protocol/Resolution/Persistence) that constrains software architecture to enable reliable LLM code generation
- A Protocol layer that enables stateful systems without breaking purity: StateGates declare reads and describe mutations without touching persistence
- Empirical demonstration across two domains: a polyglot SQL database engine and a symbolic mathematics system, with 1,123 combined database tests across JavaScript and PHP
- Evidence that the architecture is language-agnostic: identical gate structures produce identical behavior in JavaScript and PHP
- Evidence that architectural constraints enable automatic edge case handling without explicit specification

2 The SISO Framework

2.1 Core Primitives

SISO's irreducible kernel consists of three primitives in approximately 150 lines of code. These primitives are sufficient for stateless event processing and form the foundation on which the higher layers build.

Events are immutable data packets that flow through the system. Each event carries a type (its signature for dispatch) and arbitrary data:

```
class Event {
  constructor(type, data = {}) {
    this.type = type;
    this.data = data;
  }
}
```

Gates are pure transformation functions. Each gate claims one event type by signature. No two gates in a stream may share a signature—collision is a hard error, not silent precedence:

```
class Gate {
  constructor(signature) {
    this.signature = signature;
  }
  transform(event, stream) {
    // Override in subclass.
  }
}
```

Streams orchestrate event processing through registered gates. The stream maintains a signature-keyed map of gates for O(1) dispatch. Events arrive via `emit()`, the stream looks up the matching gate, and transform runs immediately (depth-first). Unclaimed events land in pending—the residue when processing settles:

```
class Stream {
  register(gate) {
    if (this.gates.has(gate.signature))
      throw new Error('Signature collision');
    this.gates.set(gate.signature, gate);
  }
  emit(event) {
    const gate = this.gates.get(event.type);
    if (gate) gate.transform(event, this);
    else this.pending.push(event);
  }
  sampleHere() {
    return { pending: [...this.pending] };
  }
}
```

StreamLog provides tiered audit observability. Streams already know, at emit-time, whether an event was claimed or fell to pending. The log records this decision at configurable levels: OFF (nothing), EVENTS (type and claimed/pending), DEEP (sub-stream lineage), and DATA (full payloads). Logging is zero-cost observation of decisions already being made.

2.2 Protocol Layer

The core primitives handle stateless transformation. Real systems require state. The Protocol layer solves this without breaking purity by splitting state access into declarations.

PureGate is a gate that takes an event and returns an event (or null). No state access, no side effects. SQL parsing, filtering, projection, and dispatch gates are PureGates. The signature is: `transform(event)` → `Event | null`.

StateGate is the critical abstraction. It splits state interaction into two pure declarations. First, `reads(event)` → `ReadSet` declares what state the gate needs—specific refs and prefix patterns. Second, `transform(event, state)` → `MutationBatch` receives the resolved state and returns a description of what should change. The gate never touches persistence. It remains a pure function of its inputs.

```
class StateGate extends Gate {
  reads(event) {
    return new ReadSet()
      .ref(`db/tables/${event.data.table}/schema`)
      .pattern(`db/tables/${event.data.table}/rows/`);
  }
  transform(event, state) {
    const rows = Object.values(state.patterns[...]);
    return new MutationBatch()
      .emit(new Event('scan_result', { rows }));
  }
}
```

ReadSet declares specific ref names and prefix patterns. **MutationBatch** describes puts (objects to store), ref sets (name-to-hash bindings), ref deletes, and follow-up events. Both are data structures, not execution—the Resolution layer interprets them.

2.3 Resolution Layer

The **Runner** is the impure shell—the only component that touches Stream, Store, and Refs. It wraps PureGates and StateGates so the Stream sees plain Gates with `transform(event, stream)`. The gates see plain events and state objects. The Runner is the quarantine boundary between purity and the world.

When a PureGate is registered, the Runner wraps it: event → gate.transform(event) → emit result. When a StateGate is registered, the Runner wraps it: event → gate.reads(event) → resolve against persistence → gate.transform(event, state) → apply mutations → emit follow-ups.

All reads flow through `Runner.resolve()`. All writes flow through `Runner.apply()`. This quarantine means every gate is testable with mock state, and the persistence mechanism can be swapped without touching any gate code.

2.4 Persistence Layer

The **Store** is a content-addressable object store. Objects are serialized to canonical JSON (sorted keys, deterministic output), hashed with SHA-256, and stored as immutable blobs. Same content always produces the same hash—deduplication is free. The store does not know what it is holding: a database row, a schema definition, a wiki page—all the same.

Refs are named references to store hashes—a flat map of strings to strings. `set(name, hash)`, `get(name) → hash | null`, `delete(name)`, `list(prefix) → [names]`. This is Git's object model: content-addressable blobs with named pointers.

The MutationBatch uses index-based indirection: `refSet(name, putIndex)` says "point this name at the object I'm putting at index 0." The Runner hashes the puts and resolves the indices. Gates compose without knowing about each other's hashes.

2.5 Architectural Constraints

SISO enforces three critical constraints:

1. Immutability. Events are immutable. Gates emit new events rather than modifying existing ones. The content-addressable store is append-only. This eliminates state-related bugs and makes each transformation independently testable.

2. Pure Functions. Gates have no side effects beyond returning events or mutation descriptions. PureGates are `event → event`. StateGates are `(event, state) → MutationBatch`. Neither touches persistence. This ensures deterministic behavior, compositional reasoning, and trivial parallelization.

3. Single Responsibility. Each gate claims one event type by signature. Complex operations are decomposed into pipelines of gates. This matches LLM capabilities—generating many simple functions is more reliable than generating one complex function.

2.6 Emergent Properties

Six properties fall out of the architecture rather than being bolted on:

- **Stateless:** Gates hold no state. PureGates see only the event. StateGates see the event and resolved state—never persistence directly.
- **Single effect:** One signature, one gate, one transform. Collision is a hard error.
- **Polyglot:** Gate/Event/Stream is a pattern, not a language feature. The Jaa database runs identically in JavaScript and PHP from the same architecture.
- **Auditable:** StreamLog observes decisions already being made. Every event records whether it was claimed or fell to pending.
- **Perfectly parallel:** No shared mutable state between gates. Streams are independent trees.
- **DOM-canonical:** For user interfaces, the DOM is the canonical state. Gates perform targeted updates—no virtual DOM, no reconciliation, no re-render cycle.

3 Empirical Evaluation

We evaluated SISO across two domains using a state-of-the-art large language model for implementation.

3.1 Jaa: SQL Database Engine

Implementation approach: Human-written domain specification provided to large language model. Dual implementation in JavaScript and PHP from the same architectural specification.

Result: ~11,600 lines across both languages implementing:

- DDL: CREATE TABLE with constraints, DROP TABLE, ALTER TABLE (add/drop column, rename), TRUNCATE
- DML: Full CRUD with complex expressions, UPSERT (ON CONFLICT), RETURNING, INSERT...SELECT
- WHERE clauses: Comparisons, AND/OR/NOT, IN, LIKE, BETWEEN, IS NULL/IS NOT NULL
- JOINs: INNER, LEFT, RIGHT, CROSS
- Subqueries: Scalar, IN, EXISTS, FROM clause
- Common Table Expressions: WITH...AS, WITH RECURSIVE
- Query features: GROUP BY, HAVING, ORDER BY, LIMIT/OFFSET, DISTINCT, UNION/UNION ALL
- Expressions: CASE WHEN, arithmetic, string functions, date/time, CAST, COALESCE, IIF
- Schema: Indexes (including UNIQUE), Views, Triggers, Constraints

- Transactions: BEGIN, COMMIT, ROLLBACK with snapshot isolation
- Persistence: Content-addressable store with SHA-256 hashing, file-backed or in-memory
- EXPLAIN for query plan inspection

Testing: 587 passing tests (JavaScript) and 536 passing tests (PHP), totaling 1,123 tests. Test coverage includes empty result sets, NULL handling in comparisons and sorting, column count validation, DEFAULT value application, constraint enforcement, transaction rollback, recursive CTEs, and edge cases (LIMIT 0, invalid syntax, duplicate tables).

Architecture: Gates organized in a pipeline. For example, SELECT processing:

```
sql -> SQLDispatchGate -> SelectParseGate -> QueryPlanGate
    -> TableScanGate -> FilterGate -> ProjectionGate
    -> OrderByGate -> LimitGate -> DistinctGate
    -> query_result (pending)
```

SQLDispatchGate and SelectParseGate are PureGates—no state needed to parse SQL. TableScanGate is a StateGate: it declares `ReadSet.pattern(`db/tables/{table}/rows/`)`, receives the resolved rows, and emits a scan_result event. InsertExecuteGate reads the schema and next_id, returns a MutationBatch with the new row and updated counter. No gate touches persistence directly.

Polyglot Validation

The JavaScript and PHP implementations share the same gate signatures, the same persistence model, and equivalent test suites. The directory structure mirrors exactly: `src/core/` maps to `ice/Core/`, `src/gates/database/` maps to `ice/Gates/Database/`, and so on. This validates that the architecture is language-agnostic: the contract is the shape, not the language.

3.2 Universal Math Gate: Symbolic Mathematics

Implementation approach: Single gate containing 216 pattern-rewrite rules across 11 mathematical domains. The stream's depth-first processing loop serves as the evaluation engine.

Result: ~600 lines implementing symbolic transformations across:

- Arithmetic: operations, factorial, identity rules
- Algebra: distribution, factoring, exponent rules, logarithms, square roots, fractions
- Calculus: derivatives (power, chain, product, quotient rules), integrals, limits
- Trigonometry: Pythagorean identities, reciprocal/quotient, sum/difference, double/half angle
- Linear Algebra: determinants, trace, transpose, dot/cross products, eigenvalues
- Set Theory: union, intersection, complement, De Morgan's laws, cardinality
- Propositional Logic: negation, conjunction, disjunction, implication, De Morgan's, distribution
- Number Theory: divisibility, GCD/LCM, modular arithmetic, congruence

- Complex Analysis: powers of i, Euler's formula, conjugates, modulus
- Differential Equations: first/second order ODEs, Laplace transforms
- Statistics: probability, Bayes' theorem, expectation, variance, standard distributions

Mechanism: An expression enters the stream. The gate pattern-matches against its rules. If a rule matches, it emits the rewritten expression, which re-enters the same gate (depth-first). When nothing matches, the expression falls to pending—that is the result. The stream's `→E→E→` loop *is* the reduction engine.

Audit trail: Every reduction step is recorded in the event's history array as `{rule, from, to}`. The derivation proof is the event chain itself—not logged by infrastructure, but carried by the data flowing through the stream.

This implementation demonstrates that CORE's event dispatch model is a term rewriting system. One gate, one stream, arbitrary mathematical depth—no special recursion mechanism required.

3.3 Edge Case Handling

In both implementations, numerous edge cases were handled correctly despite not being explicitly specified. While comprehensive edge case coverage cannot be claimed for all scenarios, the systematic nature of handling in tested cases is noteworthy.

In the database, NULL sorting correctly places NULLs last in ORDER BY (SQL standard behavior). DEFAULT values are applied when INSERT omits columns. Constraint violations raise appropriate errors. Transaction rollback correctly restores snapshot state. These patterns were not specified in prompts—the LLM inferred appropriate behavior from domain knowledge combined with SISO's architectural constraints.

4 Analysis

4.1 Why SISO Enables Reliable Generation

We identify four factors:

1. Reduced State Space. Pure functions eliminate mutable state, reducing the space of possible implementations. The Protocol layer extends this to stateful systems: StateGates declare what they need and describe what should change, but never execute mutations themselves. The state space visible to each gate is exactly what it declared in its ReadSet.

2. Compositional Structure. The gate pipeline decomposes complex operations into sequences of simple transformations. LLMs excel at generating simple functions but struggle with complex monolithic implementations. The Jaa database's SQL pipeline—parse, plan, scan, filter, project, sort, limit—is a sequence of independently simple gates.

3. Testability. Each gate is independently testable. PureGates can be tested with constructed events. StateGates can be tested with mock state objects. The Runner can be tested with in-memory persistence. This enables immediate verification during development and clear failure localization.

4. Language Agnosticism. Because the architecture is a pattern rather than a library, it transfers across languages. The same LLM prompt structure that produces a JavaScript gate produces a PHP gate. The Jaa database validates this: 587 JS tests and 536 PHP tests from the same architectural specification.

4.2 The State Problem and Its Solution

The central challenge in extending pure functional architectures to real systems is state. A SQL database must read rows, write rows, maintain schemas, and update indexes. The naive solution—allowing gates to call persistence directly—destroys purity and testability.

SISO's Protocol layer solves this by separating *declaration* from *execution*. A StateGate's `reads()` method returns a ReadSet, which is data describing what the gate needs. Its `transform()` method returns a MutationBatch, which is data describing what should change. The Runner interprets both. The gate never calls `store.get()` or `refs.set()`.

This design has a concrete benefit: the MutationBatch uses index-based indirection for ref bindings. `refSet(name, putIndex)` says "point this ref at the object I'm putting at index N." The gate does not compute hashes—the Runner does. Gates compose without coupling to each other's storage details.

4.3 Domain Applicability

SISO's effectiveness has been empirically demonstrated in deterministic domains with well-defined semantics. The framework requires that correct behavior can be specified through pure function composition.

Empirically validated domains:

- Relational databases (SQL parsing, query execution, persistence)
- Symbolic mathematics (term rewriting, algebraic simplification)

Expected to work (not yet validated):

- Web application backends (request/response processing)
- Financial systems (trading algorithms, risk analysis)
- Data pipelines and stream processing
- Compilers and interpreters for domain-specific languages
- User interfaces (DOM as canonical state, gates as targeted updaters)

May require adaptation:

- Real-time systems (timing constraints)

- Systems with significant external I/O dependencies

Unlikely to work without modification:

- Systems requiring fine-grained mutable shared state
- Domains without formal semantics
- Performance-critical inner loops (event overhead may be prohibitive)

4.4 Comparison to Traditional Development

Estimated traditional development time for equivalent implementations:

- SQL database (polyglot, with transactions, indexes, CTEs): 4–8 weeks (experienced developer)
- Symbolic math suite (11 domains): 1–2 weeks

SISO + AI achieved these in hours to days, representing 10–50x speedup. However, this assumes domain expertise in the human specifier, appropriate problem decomposition, and iterative testing and refinement.

5 Related Work

Functional Reactive Programming (FRP) shares SISO's emphasis on immutability and pure functions but focuses on time-varying values rather than discrete event transformations.

Actor Model provides message-passing concurrency but allows mutable actor state, reducing AI-compatibility.

Event Sourcing / CQRS architectures inspired SISO's event-driven design but typically include imperative command handlers. SISO's MutationBatch is a declarative alternative to imperative command execution.

Content-Addressable Storage (Git, IPFS, Nix) provides the model for SISO's persistence layer. The combination of content-addressable blobs with named refs enables deduplication, snapshot isolation, and language-agnostic storage.

LLM Code Generation: GitHub Copilot, AlphaCode, and similar tools generate code snippets but struggle with architectural consistency across large systems. SISO addresses this through enforced architectural constraints that make correct implementations the path of least resistance.

6 Limitations and Future Work

6.1 Current Limitations

- **Prototype status:** Implementations are research prototypes; production deployment requires additional validation and hardening
- **Limited validation:** Empirically validated in two domains; broader domain testing needed
- **Performance:** Event processing overhead and content-addressable hashing may impact performance-critical applications
- **Learning curve:** The Protocol/Resolution/Persistence layering adds conceptual overhead beyond the simple CORE primitives
- **Tooling:** Limited IDE support for event flow visualization and debugging

6.2 Future Directions

- Formal verification: prove gate correctness using dependent types
- Automatic optimization: fuse gates, parallelize execution
- Distributed SISO: network-transparent gate distribution
- Domain expansion: web applications, ML pipelines, user interfaces
- File-backed persistence hardening: write-ahead logging, crash recovery
- Human studies: measure learning curve and productivity

7 Conclusion

We have demonstrated that appropriate architectural constraints can enable LLMs to generate functionally complete software prototypes with minimal human intervention. The SISO framework achieves this through a layered pure functional architecture: CORE primitives for stateless event dispatch, a Protocol layer that separates state declaration from state access, a Resolution layer that quarantines impurity, and a content-addressable persistence layer.

Empirical results across two domains show 10–50x development speedup with systematic edge case handling. The Jaa database engine—1,123 tests across JavaScript and PHP from the same architecture—validates that the framework is language-agnostic. The Universal Math Gate—216 rules across 11 domains in a single gate—demonstrates that CORE's event dispatch model is a general-purpose term rewriting system.

The broader implication is significant: if architectural constraints can reliably enable AI code generation, software development may shift from implementation to specification—potentially democratizing software creation and accelerating innovation across all domains amenable to formal reasoning. However, substantial work remains to validate this approach across broader domains and at production scale.

8 Code and Data Availability

Reference implementations (Jaa database engine, Universal Math Gate), complete specifications, and comprehensive test suites are available under GPL-3.0 at: https://siso-framework.org

GitHub repository: https://github.com/siso-ai/SISO

All implementations are released under the GNU General Public License v3.0 to prevent proprietary capture and enable community development and validation.