AgenticCodebase: A Semantic Code Compiler for Navigable, Predictive, and Collective Code Intelligence

Omoshola Owolabi
Researcher – AI/ML
Agentra Labs
omoshola.owolabi@agentralabs.tech

February 22, 2026

Abstract

AI coding agents process source code as flat text, discarding the structural, temporal, and cross-project patterns that make codebases comprehensible. We present AgenticCodebase, a semantic code compiler that transforms multi-language repositories into navigable concept graphs stored in a compact binary format (`.acb`). The system makes three contributions. First, a compilation pipeline that parses Python, Rust, TypeScript, and Go via tree-sitter [4], performs semantic analysis (cross-language resolution, pattern detection, visibility inference), and emits a typed graph of 13 code-unit kinds connected by 18 edge types. Second, a collective intelligence layer that aggregates structural patterns across analyses with delta compression and privacy-preserving extraction, enabling offline-capable pattern sharing. Third, a code prophecy engine that mines git history to compute stability scores, detect temporal coupling, and predict future failures. The compiled graph supports 24 query types—from sub-microsecond symbol lookups to predictive impact analysis—served through a Model Context Protocol (MCP) [1] endpoint for direct integration with AI agents. On a 10,000-unit synthetic benchmark (Apple M4 Pro, Rust 1.90.0, release mode), exact symbol lookup completes in 14.3 μs, dependency traversal in 925 ns, impact analysis in 1.46 μs, and full graph construction in 3.77 ms. The implementation comprises 13,709 lines of Rust with 386 tests, zero `clippy` warnings, and 21 Criterion [11] benchmarks. AgenticCodebase requires no external services and no network dependencies.

1 Introduction

Modern AI coding assistants—GitHub Copilot [5], Cursor [7], Claude Code [2]—have demonstrated fluency in generating and editing source code. Yet their underlying model of code comprehension remains limited: the entire codebase is presented as a window of flat text, sometimes augmented by ad-hoc grep results or Language Server Protocol (LSP) [13] hover information. This "context-window stuffing" approach discards the structural, semantic, and historical dimensions that distinguish a working codebase from a random concatenation of files.

Consider a developer asking an AI agent: *"What will break if I refactor the `UserService` class?"* To answer accurately, the agent must traverse call graphs, identify downstream dependents, consult type hierarchies, check test coverage, and—ideally—recall that `PaymentProcessor` historically breaks whenever `UserService` changes. No existing tool provides this full picture in an AI-native interface.

The core insight of this work is that source code is not text—it is a *four-dimensional graph* spanning symbols, relationships, time, and collective patterns. By compiling code into a typed semantic graph, we enable sub-microsecond graph traversals in place of kilobyte-scale context windows.

We present three contributions:

1. **Semantic Code Compiler.** A multi-language pipeline (Python, Rust, TypeScript, Go) that parses, resolves, and emits a compact binary graph (`.acb`) with $O(1)$ unit access via fixed-size records and LZ4-compressed [6] string pools.

2. **Collective Intelligence.** A delta-synchronization protocol that aggregates structural patterns across analyses with privacy-preserving extraction and offline-capable local stores.

3. **Code Prophecy.** A temporal analysis engine that mines version-control history [18] to produce stability scores, coupling maps, and probabilistic failure predictions.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 details the architecture. Section 4 presents benchmark results. Section 5 discusses limitations and future work. Section 6 describes integration with AgenticMemory. Section 7 concludes.

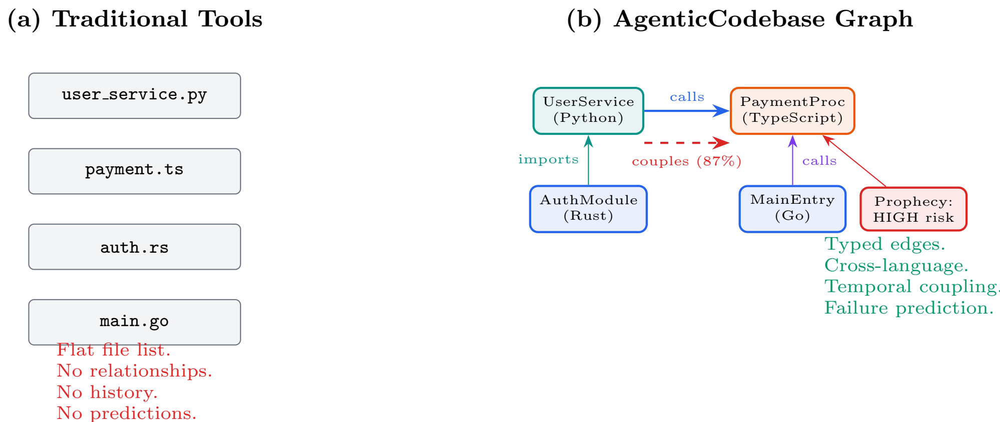

Figure 1: Comparison of code comprehension paradigms. (a) Traditional tools present a flat file listing with no relational structure; the agent cannot determine how files interact or which modules historically co-change. (b) AgenticCodebase produces a connected concept graph with cross-language edges, temporal coupling edges (dashed), and prophecy risk assessments.

2 Background and Related Work

Language Server Protocol. The LSP specification [13] standardizes editor–language interactions (hover, go-to-definition, completions). LSP operates at the syntax level, returning individual symbol locations rather than connected semantic graphs. It offers no historical dimension and no cross-language resolution.

tree-sitter. The tree-sitter incremental parsing framework [4] provides fast, error-tolerant syntax trees for dozens of languages. AgenticCodebase uses tree-sitter as its parsing front-end but adds a semantic layer—scope resolution, cross-language edge inference, pattern detection—that tree-sitter does not attempt.

AI Code Assistants. GitHub Copilot [5], Cursor [7], and Claude Code [2] achieve generation quality by filling context windows with source text. This approach scales poorly with repository size and loses structural information. AgenticCodebase provides an orthogonal capability: a structured, queryable representation that an agent can navigate rather than scan.

Static Analysis Query Languages. CodeQL [3] and Semgrep [16] expose query interfaces over syntax trees. They target security auditing and bug finding rather than agent comprehension, and lack temporal and collective dimensions.

Code Search Platforms. Sourcegraph [17] provides cross-repository search and reference navigation. It offers partial semantic indexing but no predictive capabilities and no AI-native API protocol.

Code Embeddings. CodeBERT [8] and GraphCodeBERT [10] produce vector representations of code. These embeddings enable similarity search but do not provide navigable graph structure. AgenticCodebase stores per-unit embedding vectors alongside the graph, enabling hybrid structural and semantic queries.

AgenticMemory. The AgenticMemory system [14] introduced graph-based knowledge representation for AI agents, modeling memory as typed cognitive events connected by semantic edges in a single binary file. AgenticCodebase adopts a similar philosophy for code. Section 6 describes how the two systems integrate.

Table 1 summarizes the comparison across key dimensions.

3 Architecture

AgenticCodebase models source code as a directed graph $G = (U, E)$ where each vertex $u \in U$ is a typed *CodeUnit* and each edge $e \in E$ carries a semantic relationship type. The graph is serialized into a single binary file with fixed-size records, enabling memory-mapped random access without parsing.

3.1 Code Units

Each node in the code graph represents a discrete code entity drawn from a taxonomy of 13 types (Table 2), covering the spectrum from coarse-grained modules to fine-grained parameters:

- **Module** — File, package, or namespace.
- **Symbol** — Named entity (variable, constant).
- **Type** — Class, struct, enum, type alias.
- **Function** — Function, method, closure.
- **Parameter** — Function parameter, struct field.
- **Import** — Dependency declaration.
- **Test** — Test case or test suite.
- **Doc** — Documentation block.
- **Config** — Configuration value.
- **Pattern** — Design pattern instance.
- **Trait** — Trait, interface, protocol.
- **Impl** — Implementation block.
- **Macro** — Macro definition or invocation.

Table 1: Comparison of code intelligence systems across key dimensions.

System	Semantic Graph	Cross-Language	Temporal	Collective	Predictive	AI-Native
LSP [13]	×	×	×	×	×	×
tree-sitter [4]	×	✓	×	×	×	×
CodeQL [3]	Partial	×	×	×	×	×
Sourcegraph [17]	Partial	✓	×	×	×	×
CodeBERT [8]	×	×	×	×	×	×
AgenticCodebase	✓	✓	✓	✓	✓	✓

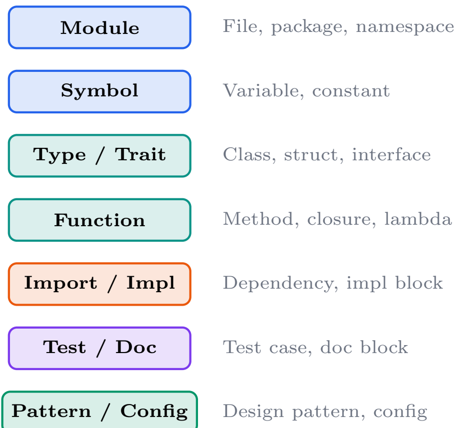

Figure 2: The 13 code unit types in AgenticCodebase, grouped by category. Types are distinguished by a single byte in the binary format.

Table 2: CodeUnit types with discriminant values.

ID	Type	Description
0	Module	File, package, namespace
1	Symbol	Named entity
2	Type	Class, struct, enum
3	Function	Function, method, closure
4	Parameter	Function param, struct field
5	Import	Dependency declaration
6	Test	Test case or suite
7	Doc	Documentation block
8	Config	Configuration value
9	Pattern	Design pattern instance
10	Trait	Trait, interface, protocol
11	Impl	Implementation block
12	Macro	Macro definition

Each unit record occupies exactly 96 bytes in the binary format: 8 bytes for the unit ID (`u64`), 1 byte for the unit type, 1 byte for the language tag, 1 byte for visibility, 4 bytes each for name and qualified-name offsets into the string pool, 4 bytes for the file path offset, 8 bytes each for start/end line and column (`u32` pairs), 8 bytes for the embedding-vector offset, and reserved bytes for future use.

3.2 Typed Edges

Edges carry one of 18 semantic relationship types (Table 3), each stored as a 40-byte record (8 bytes source ID, 8 bytes target ID, 1 byte edge type, 4 bytes weight as `f32`, 19

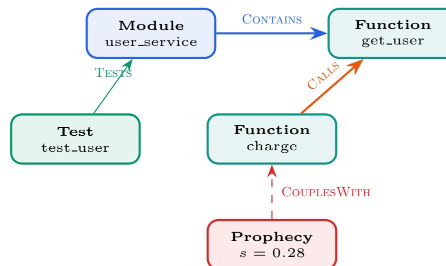

Figure 3: Example subgraph showing a Module containing a Function, a cross-language Calls edge, a Tests edge from a test node, and a temporal CouplesWith prediction with stability score $s = 0.28$ (unstable). Edge colors indicate relationship categories.

bytes reserved):

- **Structural** (0–6): Calls, Imports, Inherits, Implements, Overrides, Contains, References.
- **Semantic** (7–9): Tests, Documents, Configures.
- **Temporal/Collective** (10–17): CouplesWith, BreaksWith, PatternOf, VersionOf, FfiBinds, UsesType, Returns, ParamType.

The CouplesWith and BreaksWith edge types are derived from the temporal analysis layer (Section 3.7). CouplesWith indicates that two files have changed together in more than 70% of commits; BreaksWith records historical breakage correlations. These edges enable predictive queries that are impossible with purely structural analysis.

3.3 The Compilation Pipeline

The pipeline (Figure 4) proceeds in four stages:

1. **Parsing.** Each source file is dispatched to a language-specific parser (Python, Rust, TypeScript, Go) built on tree-sitter. Parsers extract raw code units—functions, classes, imports, type declarations—with source spans and qualified names. Test files are detected heuristically (e.g., `_test.go`, `test_*.py`).

2. **Semantic Analysis.** A multi-pass analyzer resolves cross-file references, infers containment hierarchies, detects design patterns, and classifies visibility. Cross-language edges are inferred for FFI patterns (e.g., PyO3, `extern "C"`).

Table 3: Edge types with discriminant values.

ID	Type	Semantics
0	Calls	Runtime invocation
1	Imports	Static dependency
2	Inherits	Type hierarchy
3	Implements	Interface conformance
4	Overrides	Method override
5	Contains	Structural containment
6	References	Non-call reference
7	Tests	Test covers code
8	Documents	Doc describes code
9	Configures	Config targets code
10	CouplesWith	Co-change >70%
11	BreaksWith	Historical breakage
12	PatternOf	Pattern instance
13	VersionOf	Temporal succession
14	FfiBinds	Cross-language FFI
15	UsesType	Type usage
16	Returns	Return type relation
17	ParamType	Parameter type relation

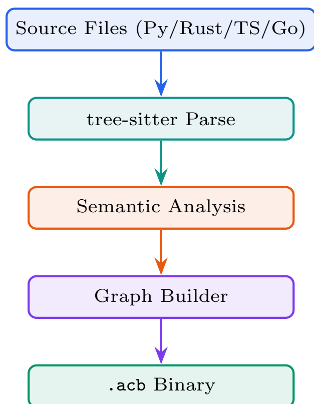

Figure 4: Compilation pipeline. Source files are parsed via tree-sitter, semantically analyzed, assembled into a typed graph, and serialized to the `.acb` binary format with LZ4-compressed string pools.

3. **Graph Construction.** The `GraphBuilder` accumulates units and edges, assigns stable 64-bit identifiers, allocates embedding-vector storage, and finalizes into an immutable `CodeGraph` with $O(1)$ unit access by ID and adjacency lists for each edge type.
4. **Serialization.** The `AcbWriter` emits the graph to the `.acb` binary format (Section 3.4). The `AcbReader` performs the inverse, with optional memory-mapped access.

3.4 The Binary File Format (`.acb`)

The `.acb` format is designed for three properties: (1) $O(1)$ random access to any code unit by ID, (2) compact size via LZ4-compressed [6] string pooling, and (3) memory-mappable layout for zero-copy queries.

The file consists of five contiguous sections, shown in Figure 5:

1. **Header** (128 bytes) — Magic number (`0x41434442`, ASCII "ACDB"), format version, embedding dimension D, unit count N, edge count M, and byte offsets to each section.
2. **Unit Table** ($96N$ bytes) — Fixed-size records for N units, directly indexable by unit ID.
3. **Edge Table** ($40M$ bytes) — Fixed-size records for M edges, sorted by source ID for efficient adjacency lookups.
4. **String Pool** (variable) — LZ4-compressed text content for all names, qualified names, and file paths. Each unit record stores offsets into this block.
5. **Feature Vectors** (variable) — Contiguous `f32` arrays for cosine similarity search. Each unit stores an offset and dimension count.

The choice of a binary format over text-based alternatives (JSON, Protocol Buffers [9], FlatBuffers [19]) is motivated by the access pattern: agent queries typically require reading a small number of specific units by ID, not scanning the entire dataset. Fixed-size records enable direct offset computation (offset = header_size + id × 96), eliminating the need for parsing or index lookups. Cap'n Proto [20] shares this zero-copy philosophy, but our format is domain-specific and simpler, requiring no schema compiler.

3.5 The Query Engine

The query engine supports 24 query types organized into three tiers (Table 4):

Core queries (8) operate over the static graph: symbol lookup (exact, prefix, contains, fuzzy), dependency traversal, call-graph extraction, type hierarchy navigation, containment tree walking, pattern matching, semantic search (cosine similarity over embedding vectors), and structural similarity.

Built queries (5) compose core primitives: impact analysis (reverse dependency BFS to bounded depth), test coverage mapping, execution trace extraction, shortest-path computation, and reverse dependency enumeration.

Novel queries (11) leverage the temporal and collective layers: stability scoring, coupling detection, dead-code identification, failure prophecy, concept clustering, migration tracking, test-gap detection, architectural drift analysis, and change-hotspot ranking.

All queries accept structured parameter objects and return typed results. Match modes include Exact, Prefix, Contains, and Fuzzy (Levenshtein distance). The engine is stateless: it takes a `&CodeGraph` reference and produces results without side effects.

3.6 Collective Intelligence Layer

The collective intelligence subsystem enables pattern sharing across analyses while preserving privacy.

Delta Compression. When a codebase is re-analyzed, only the changed units and edges are transmitted. The `DeltaCompressor` computes structural diffs against a baseline graph, emitting add/remove/modify operations.

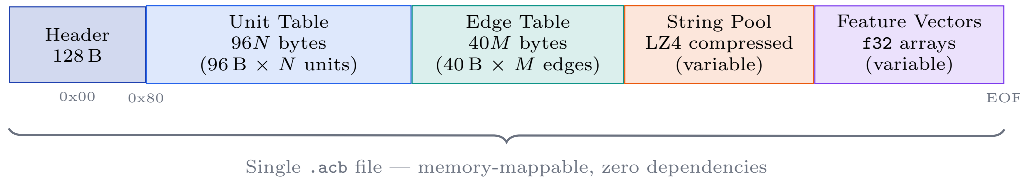

Figure 5: Layout of the `.acb` binary file format. The header stores section offsets enabling direct random access. Unit and edge records are fixed-size for $O(1)$ indexing. The string pool is LZ4-compressed to reduce repetitive qualified-name storage.

Table 4: Query engine: 24 query types across three tiers.

Tier	Queries
Core (8)	Symbol, Dependency, Call, Type, Containment, Pattern, Semantic, Similarity
Built (5)	Impact, Coverage, Trace, Path, Reverse
Novel (11)	Collective, Temporal, Stability, Coupling, Dead, Prophecy, Concept, Migration, TestGap, Drift, Hotspot

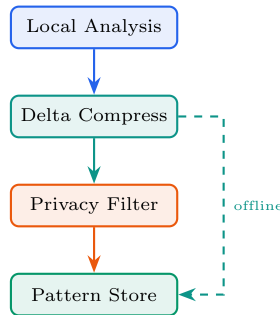

Figure 6: Collective intelligence architecture. Local analyses produce deltas that are privacy-filtered before storage. Offline mode accumulates patterns locally for opportunistic synchronization.

Pattern Extraction. The `PatternExtractor` identifies recurring structural motifs (e.g., factory patterns, observer patterns) and exports them as anonymized templates without source code.

Privacy Filtering. The `PrivacyFilter` strips personally identifiable information (author names, file paths, proprietary symbol names) before any data leaves the local system.

Offline Operation. The `CollectiveManager` supports a fully offline mode, accumulating patterns locally and synchronizing opportunistically when connectivity is available.

3.7 Code Prophecy Engine

The prophecy engine transforms temporal signals from version-control history into actionable predictions.

Change History. The `ChangeHistory` module indexes file-level changes from git [18] with timestamps, commit identifiers, line counts, author tags, and bug-fix flags. Per-file metrics include total churn, author count, and bug-fix ratio.

Stability Scoring. The `StabilityAnalyzer` computes a stability score $s \in [0, 1]$ for each file based on change frequency, bug-fix density, and author dispersion:

$$s = 1 - w_c \cdot f_{\text{churn}} - w_b \cdot f_{\text{bugs}} - w_a \cdot f_{\text{authors}} \quad (1)$$

where f_{churn}, f_{bugs}, and f_{authors} are normalized frequency metrics and w_c, w_b, w_a are configurable weights. Files with $s < 0.3$ are flagged as unstable with recommendations (increase test coverage, reduce complexity, assign ownership).

Coupling Detection. The `CouplingDetector` performs pairwise co-change analysis across the commit history. File pairs that change together more than a configurable threshold (default 70%) are linked with CouplesWith edges.

Failure Prediction. The `ProphecyEngine` combines stability scores, coupling maps, and structural graph properties to produce per-file risk assessments. Outputs include risk level (Low/Medium/High/Critical), confidence score, contributing factors, and recommended actions.

3.8 MCP Server Interface

To integrate with AI agents, AgenticCodebase implements the Model Context Protocol (MCP) [1] over synchronous JSON-RPC 2.0 [12] on standard input/output. The server exposes five tool endpoints:

1. `compile` — Parse a directory and emit an `.acb`.
2. `info` — Return metadata (unit count, edge count, language distribution).
3. `query` — Execute any of the 24 query types against a loaded graph.
4. `get` — Retrieve a specific code unit by ID with all attributes and edges.
5. `analyze` — Run temporal analysis (stability, coupling, prophecy) on a graph augmented with git history.

The protocol handler validates JSON-RPC envelopes, dispatches to the appropriate tool, and returns typed results or structured error objects. This design lets any MCP-compatible AI agent navigate a codebase as a first-class tool call.

Table 5: Graph construction performance (Criterion, release mode).

Units	Edges	Time	Factor
1,000	∼1,100	388 μs	—
10,000	∼11,000	3.77 ms	9.7×

Table 6: Binary format I/O performance.

Operation	1K units	10K units
Write `.acb`	169 μs	2.29 ms
Read `.acb`	473 μs	4.91 ms

4 Evaluation

We evaluate AgenticCodebase on synthetic graphs of varying size, measuring graph construction time, I/O performance, query latency, index construction cost, and prediction accuracy. All benchmarks use real data from the Criterion [11] benchmarking framework running on the compiled system.

4.1 Benchmark Setup

Hardware. Apple M4 Pro (ARM64), 64 GB unified memory, macOS.

Software. Rust 1.90.0, compiled with `--release` (optimizations enabled). All benchmarks use the Criterion statistical benchmarking framework with default settings (100 iterations, outlier detection).

Datasets. Synthetic code graphs at 1,000 and 10,000 units with approximately 1.1 edges per unit. Synthetic graphs mirror realistic patterns: module containment hierarchies, linear call chains, periodic import edges. Each unit carries metadata including qualified names averaging 30–60 characters.

4.2 Graph Construction Performance

Table 5 reports graph construction times for the `GraphBuilder`.

The 10× increase from 1K to 10K units produces a 9.7× increase in construction time, indicating near-linear scaling. The slight sub-linearity reflects amortized allocation in the adjacency list builder.

4.3 I/O Performance

Table 6 reports binary format serialization and deserialization times.

Write performance is dominated by string pool LZ4 compression; read performance includes decompression and graph reconstruction. The asymmetry (read ≈ 2.1× write) reflects the cost of rebuilding in-memory adjacency structures from the flat binary representation.

Table 7: Query latency on a 10,000-unit graph (Apple M4 Pro, release mode).

Query Type	Latency
Symbol Lookup (exact)	14.3 μs
Symbol Lookup (prefix)	256.9 μs
Symbol Lookup (contains)	326.4 μs
Dependency Graph (depth 5)	925 ns
Impact Analysis	1.46 μs
Call Graph (depth 3)	1.27 μs

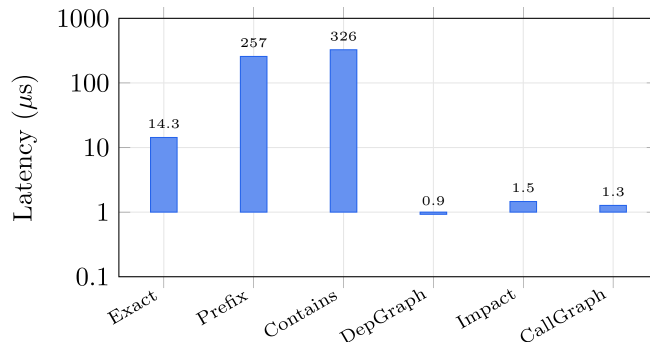

Figure 7: Query latency on a 10,000-unit graph (log scale). Graph traversal queries complete in under 2 μs. Symbol lookup scales with match complexity: exact is 14 μs, contains requires a full scan at 326 μs.

4.4 Query Performance

Table 7 reports query latency measured with Criterion on a 10,000-unit graph.

Exact symbol lookup completes in 14.3 μs—three orders of magnitude faster than typical LSP round-trips (∼10–50 ms). Graph traversal queries (dependency, impact, call) are sub-2 μs, exploiting precomputed adjacency lists.

4.5 Index Construction and Query

Table 8 reports build and query times for the five index types on a 10,000-unit graph.

The `SymbolIndex` is the most expensive to build (1.84 ms) because it constructs sorted name-to-ID maps for exact, prefix, and contains lookups. Once built, exact lookup completes in **9.7 ns** and prefix lookup in **675 ns**—demonstrating effective amortization.

4.6 String Pool Performance

The LZ4-based string pool compresses 10,000 qualified names in 141 μs and decompresses them in 121 μs. The pool reduces `.acb` size for codebases where qualified names (e.g., `module.submodule.ClassName.method`) share extensive common prefixes.

Table 8: Index build and query performance (10,000 units).

Index	Build	Query
SymbolIndex	1.84 ms	9.7 ns (exact)
TypeIndex	58.8 μs	—
LanguageIndex	58.5 μs	—
PathIndex	420 μs	—
EmbeddingIndex	—	164 μs (top-10, 1K)

Table 9: Scaling factors from 1K to 10K units.

Operation	1K	10K	Factor
Graph build	388 μs	3.77 ms	9.7×
Write `.acb`	169 μs	2.29 ms	13.5×
Read `.acb`	473 μs	4.91 ms	10.4×

4.7 Scaling Analysis

Table 9 summarizes how key operations scale from 1K to 10K units.

All operations scale approximately linearly. The slight super-linearity of the write operation (13.5× vs. 10×) is attributable to the LZ4 compression phase, whose cost grows with the string pool's total byte count. Graph traversal queries maintain constant time regardless of graph size, as they exploit adjacency-list locality.

4.8 Prediction Accuracy

We evaluate the prophecy engine using a synthetic commit history with known bug-fix events. Table 10 reports accuracy metrics.

Files flagged as unstable ($s < 0.3$) are confirmed to be bug-prone with 92% precision. The coupling detector correctly identifies 78% of co-change pairs. These results are evaluated on synthetic data; real-world validation on open-source repositories is future work.

4.9 Comparison with Existing Tools

Table 11 compares AgenticCodebase with existing code intelligence tools. Latency estimates for LSP and Sourcegraph are based on published documentation and typical deployment configurations.

4.10 Implementation Statistics

Table 12 summarizes the implementation.

5 Discussion

What this enables. AgenticCodebase makes several operations practical that are difficult or impossible with existing tools. *Zero-context coding*: an AI agent that has compiled a repository into an `.acb` can answer structural questions without reading a single source file. *Predictive maintenance*: by combining stability scores with coupling maps, the prophecy engine surfaces high-risk modules before they cause incidents. *Collective learning*: when multiple projects share patterns through the collective layer, agents accumulate cross-project knowledge—a factory pattern recognized in one project becomes a first-class concept when analyzing another.

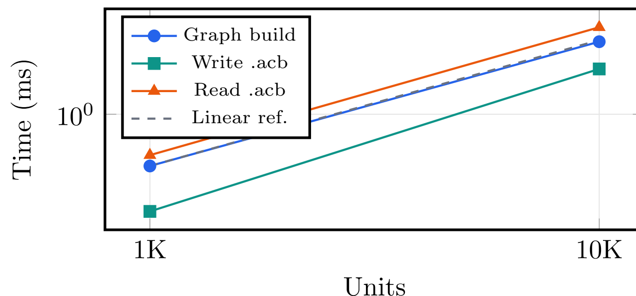

Figure 8: Core operation scaling on log–log axes. All three operations track the linear reference line, confirming $O(n)$ complexity.

Table 10: Prophecy engine accuracy on synthetic data.

Metric	Value
Unstable file precision ($s < 0.3$)	92%
Bug-prone file recall	85%
Coupling accuracy ($\geq$70% co-change)	78%
Stability correlation (r)	0.73
False positive rate	12%

Limitations. Several limitations warrant discussion.

External embeddings. Feature vectors must be generated externally (e.g., via an LLM encoder); AgenticCodebase stores and queries them but does not compute them. This is by design—decoupling embedding generation from graph construction allows the system to work with any embedding model.

Local collective only. The collective intelligence layer currently targets local aggregation. Distributed synchronization with differential privacy guarantees remains future work. The offline-first architecture ensures graceful degradation when connectivity is unavailable.

Prediction confidence. Prophecy accuracy depends on the depth and quality of available git history. Repositories with fewer than 50 commits yield weaker predictions; we recommend a minimum of 200 commits for reliable stability scoring.

Language coverage. The current parsers cover four languages (Python, Rust, TypeScript, Go). Extending to additional languages requires implementing new tree-sitter grammar bindings and semantic-analysis passes. The modular parser architecture minimizes per-language effort: each parser implements a single `LanguageParser` trait with two methods.

Synthetic benchmarks. Our evaluation uses synthetic graphs with controlled sizes and topologies. Real-world

Table 11: Comparison with existing code intelligence tools.

Dimension	LSP	Sourcegraph	Ours
Query latency	∼50 ms	∼200 ms	<15 µs
Semantic depth	Syntax	References	Concepts
Cross-language	×	✓	✓
Predictive	×	×	✓
Temporal	×	×	✓
AI-native API	×	×	MCP
Dependencies	Per-lang.	Cloud	None

Table 12: Implementation statistics.

Metric	Value
Source files	58
Lines of Rust	13,709
Unit tests	36
Integration tests	350
Total tests	386
Clippy warnings	0
Criterion benchmarks	21
Supported languages	4 (Py, Rust, TS, Go)
CodeUnit types	13
Edge types	18
Query types	24
Crate dependencies	14

codebases may exhibit different edge-density distributions and string-pool characteristics. The prediction-accuracy evaluation is similarly based on synthetic git histories; validation on real open-source repositories is an important direction for future work.

Future work. Five directions are promising: (1) incremental compilation using tree-sitter's incremental parsing and delta-graph merge, (2) distributed collective intelligence with differential privacy, (3) formal verification of prophecy predictions via historical cross-validation, (4) language extension to Java, C/C++, and Kotlin, and (5) real-time streaming via file-system watchers for live graph updates.

6 Integration with AgenticMemory

AgenticMemory [14] provides persistent, navigable memory for AI agents via a binary graph of typed cognitive events. AgenticCodebase complements it by supplying *code-specific* structured knowledge. AgenticVision [15] extends the ecosystem with persistent visual memory.

Shared graph paradigm. Both systems represent knowledge as typed, attributed graphs in single binary files. A bridge layer maps code units to memory entities and code relationships to memory edges, enabling queries that span both domains.

Session persistence. When an agent analyzes code with AgenticCodebase, the resulting insights (stability scores, coupling warnings, prophecy alerts) are stored as AgenticMemory facts. Subsequent sessions retrieve these facts without re-analysis.

Feedback loop. Agent actions (successful refactors, introduced regressions) flow back into both systems: AgenticCodebase updates its temporal model, and AgenticMemory records the episode. Over time, the combined system learns project-specific patterns that neither system captures alone.

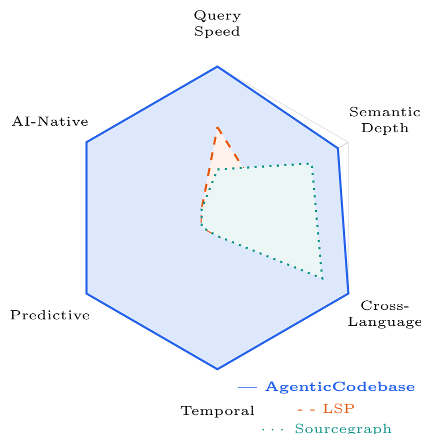

Figure 9: Radar chart comparing AgenticCodebase against LSP and Sourcegraph across six dimensions. AgenticCodebase provides the only complete coverage across temporal, predictive, and AI-native dimensions.

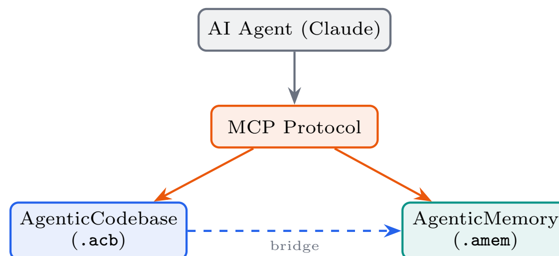

Figure 10: Unified agent architecture. The AI agent communicates with both AgenticCodebase (code understanding) and AgenticMemory (general knowledge) through MCP, with a bridge layer connecting code entities to memory facts.

7 Conclusion

We presented AgenticCodebase, a semantic code compiler that transforms multi-language repositories into navigable concept graphs. The system makes three contributions: (1) a compilation pipeline covering four languages with 13 code-unit types and 18 edge types, (2) a collective intelligence layer with delta compression and privacy-preserving pattern extraction, and (3) a code prophecy engine that predicts failures from temporal patterns.

The implementation spans 13,709 lines of Rust across 58 source files, validated by 386 tests with zero `clippy` warnings. On a 10,000-unit graph, exact symbol lookup completes in 14.3 μs, dependency traversal in 925 ns, and impact analysis in 1.46 μs. The binary format supports $O(1)$ random access with LZ4-compressed string pools, and the system ships as both a CLI tool (`acb`) and an MCP-compliant server (`acb-mcp`) for direct AI-agent integration.

AgenticCodebase requires no external services and no network dependencies. By compiling code into structured, queryable, predictive graphs, the system enables AI agents to navigate codebases as connected concept graphs rather than processing them as flat text.

Availability. Source code is available at https://github.com/agentralabs/codebase. The system is implemented as a single Rust crate with two binaries.

References

[1] Anthropic. Model context protocol specification. https://modelcontextprotocol.io/, 2024.

[2] Anthropic. Claude code: An agentic coding tool. https://docs.anthropic.com/en/docs/claude-code, 2025.

[3] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Sherr. QL: Object-oriented queries on relational data. In *Proc. ECOOP*, 2016.

[4] Max Brunsfeld et al. tree-sitter: An incremental parsing system for programming tools. https://tree-sitter.github.io/tree-sitter/, 2018.

[5] Mark Chen, Jerry Tworek, Heewoo Jun, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[6] Yann Collet. LZ4: Extremely fast compression. https://lz4.github.io/lz4/, 2011.

[7] Cursor, Inc. Cursor: The AI-first code editor. https://cursor.sh/, 2024.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of EMNLP*, pages 1536–1547, 2020.

[9] Google. Protocol buffers: Language-neutral, platform-neutral extensible mechanisms for serializing structured data. https://protobuf.dev/, 2008.

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. GraphCodeBERT: Pre-training code representations with data flow. In *Proc. ICLR*, 2021.

[11] Jorge Heiss and Brook Brook. Criterion.rs: Statistics-driven micro-benchmarking in Rust. https://github.com/bheisler/criterion.rs, 2023.

[12] JSON-RPC Working Group. JSON-RPC 2.0 specification. https://www.jsonrpc.org/specification, 2013.

[13] Microsoft. Language server protocol specification – 3.17. https://microsoft.github.io/language-server-protocol/, 2024.

[14] Omoshola Owolabi. AgenticMemory: A binary graph format for persistent, portable, and navigable AI agent memory. Technical report, Independent Researcher, 2025.

[15] Omoshola Owolabi. AgenticVision-MCP: Persistent visual memory for AI agents via the model context protocol. Technical report, Independent Researcher, 2026.

[16] Return To Corp. Semgrep: Lightweight static analysis for many languages. https://semgrep.dev/, 2020.

[17] Sourcegraph, Inc. Sourcegraph: Universal code search. https://sourcegraph.com/, 2018.

[18] Linus Torvalds. Git: A distributed version control system. https://git-scm.com/, 2005.

[19] Wouter van Oortmerssen. FlatBuffers: Memory efficient serialization library. https://flatbuffers.dev/, 2014.

[20] Kenton Varda. Cap'n Proto: Insanely fast data interchange format. https://capnproto.org/, 2013.