Title

Systems and Methods for Secure Bottom-Up Comprehension, Maintenance, and Evolution of Software Codebases via Sandboxed Autonomous Agents

---

Abstract

Disclosed herein are systems and methods for enabling autonomous agents to securely comprehend, document, maintain, and evolve complex software codebases using bottom-up dependency analysis, structured memory, deterministic sandboxing, and coordinated agent swarms. Unlike top-down AI assistants focused on reactive code suggestions, this architecture supports persistent, auditable workflows—including test generation, invariant inference, secure sandbox simulation, human-readable documentation, and even articles and graphics for popular consumption. The system integrates seamlessly alongside tools such as Copilot and Cursor, yet overcomes their enterprise adoption barriers by offering verifiable execution, reproducibility, and compliance with governance policies. Key features include cryptographic commit signing, structured tool orchestration, blockchain-aware testing, and natural language interfaces for non-technical users.

Field of the Invention

This invention relates to the fields of software engineering, artificial intelligence, and secure system orchestration, particularly systems for bottom-up comprehension, collaborative agent reasoning, and secure sandboxed execution in support of software maintenance and improvement at scale.

Background

Enterprises managing large-scale codebases face growing challenges: onboarding new developers, avoiding regressions, maintaining documentation, ensuring test coverage, and enforcing compliance. Traditional AI tools like Copilot and Cursor assist individual developers but lack structured comprehension, sandboxed validation, and persistent memory. They offer convenience, but not correctness.

By contrast, organizations require systems that:

- Understand the codebase holistically;
- Evolve over time with traceable reasoning;
- Interact safely with production-adjacent data and infrastructure;
- And empower non-engineers without compromising software integrity.

Today, there is no system that reliably combines automated reasoning, secure execution, auditable workflows, and collaborative interfaces for both technical and non-technical users.

Summary of the Invention

The disclosed system enables organizations to scale software comprehension and maintenance through a secure, agent-driven architecture featuring:

- **Bottom-up comprehension** via dependency graph traversal, beginning with leaf functions and inductively grokking upstream modules.
- **Agent-generated tests and documentation**, including preconditions, postconditions, invariants, and orthogonal unit tests validated by simulation.
- **Deterministic sandbox environments**, persistent across sessions, enabling function execution, fuzz testing, and multi-user workflow simulation without production risk.
- **Structured memory orchestration**, supporting durable reasoning, audit trails, and reproducible experiments.
- **Tool-based harness interface**, where agents issue structured commands such as `commit()`, `http()`, `execute()`, `rollback()`, and receive structured results (logs, errors, outputs).
- **Version control integration**, where pull requests are auto-generated, signed, and submitted after passing dynamic test quorums.
- **Private key storage**, enabling secure commit signing and testnet blockchain interactions without key exposure.

- **Cryptographic upgrade governance**, requiring M-of-N auditor signatures for any sandbox evolution or policy override.
- **Swarm-based reasoning**, where multiple agents operate in parallel, cover disjoint scopes, and coordinate integration test generation.
- **Natural language interfaces**, empowering business users to propose changes, trigger tests, and understand system behavior in plain English.

This architecture transforms the role of AI from reactive assistant to proactive steward—bridging the gap between developer tools and enterprise-grade automation.

Detailed Description:

---

1. Bottom-Up Dependency Grokking

Traditional AI coding assistants operate top-down: they generate snippets in response to user prompts but lack persistent understanding of the broader system. This invention introduces a bottom-up approach, where comprehension is built inductively from leaf-level functions through dependency graphs. Each function is analyzed in isolation, allowing agents to build composable, testable units of knowledge, which are later combined to understand larger components. This approach forms the backbone of the system’s reliability and reproducibility.

The system:

- Parses the software codebase into an intermediate representation, such as an abstract syntax tree (AST) or call graph.
- Identifies leaf nodes — functions or modules with no internal dependencies — as starting points.
- Assigns agents to analyze each leaf node individually, extracting inputs, outputs, side effects, and invariants.
- Combines verified knowledge of dependencies to understand higher-level functions inductively.
- Stores the results in structured memory, enabling future queries, audits, and reasoning.

Benefits:

- Avoids hallucinations common in top-down AI systems.
- Enables structured, composable understanding that scales.
- Reduces onboarding time and regression risk for enterprise teams.

---

2. Autonomous Documentation, Testing, and Invariants

Modern codebases often lack structured documentation or reliable tests. This system uses AI agents to extract specifications and generate human-readable documentation and orthogonal test cases for each function. These assets are versioned and validated automatically, building confidence in code behavior.

- For each understood code unit, the system generates:
  - **Preconditions** (required inputs)
  - **Postconditions** (guaranteed outputs)
  - **Invariants** (persisting truths during execution)
  - **Unit tests** targeting specific behaviors
  - **Human-readable documentation** and even articles and graphics for publishing
- Unit tests undergo **orthogonality validation**: if a minor change breaks only one test while others pass, the test is considered effective and non-redundant.

Benefits:

- Ensures deeper, contract-driven code quality.
- Reduces developer maintenance workload.
- Improves auditability and compliance readiness.

---

3. Secure Sandboxed Execution

Unlike Copilot or Replit-style execution, this system ensures that all code runs in a deterministic, persistent sandbox that cannot affect production. The sandbox is tamper-proof, governed by quorum-controlled scripts, and suitable for sensitive industries requiring HIPAA or PCI compliance.

- All agent analysis and testing occurs inside isolated, deterministically seeded sandboxes.
- Sandboxes encrypt all data at rest.
- Updates to the sandbox code or environment can occur only via scripts pulled internally, validated by quorum-signed approvals.
- Sandboxes simulate user workflows, multi-user scenarios, time progression, blockchain transaction submissions, and fuzz testing.

Benefits:

- Prevents accidental or malicious disruption of production environments.
- Enables reproducible experiments for developers and auditors.
- Allows safe experimentation without compliance risk.

---

4. Structured Memory and Harness Tool Use

To maintain determinism and auditability, the system replaces direct tool use with a harness that interprets structured instructions like `commit()`, `http()`, and `rollback()`. This makes all agent behavior reproducible and traceable, and enables future debugging or external verification.

- Agents interact with a **harness** that interprets structured tool instructions:
  - **store()**, **retrieve()**, **commit()**, **rollback()**, **http()**, **execute function()**

- Results such as compiler errors, runtime logs, and HTTP responses are returned in structured formats.
- The system uses **Retrieval-Augmented Generation (RAG)** principles internally, without relying on vector databases.

Benefits:

- Standardizes agent-harness communication.
- Enables deterministic and verifiable tool execution.
- Simplifies audits and debugging.

---

5. Persistent Codebase Maintenance and Improvement

Once comprehension is achieved, agents continuously monitor for changes in the codebase. When new commits arrive, agents evaluate them against known invariants, revalidate tests, and propose bugfixes or improvements—creating a persistent virtual engineering team.

- Agents monitor version control repositories.
- When changes occur:
    - Validate modifications against known behavior and invariants.
    - Detect regressions, propose bugfixes, submit pull requests.
    - Generate additional documentation and tests as needed.
- Bugfix agents are activated if failures are detected during testing.

Benefits:

- Acts as a persistent virtual senior engineering team.
- Reduces the risk of undetected regressions.

- Saves corporations significant developer time and cost.

---

6. Agent Swarms Operating in Parallel

To handle large codebases, the system deploys multiple agents in parallel, each handling a disjoint scope of the dependency graph. This approach enables massive horizontal scaling and specialization—e.g., one agent for fuzzing, one for test generation, one for doc writing.

- The system divides the codebase dependency graph into scopes.
- Multiple agents operate in parallel across non-overlapping scopes.
- Agents grok their scope independently and contribute findings to structured memory.
- Integration agents coordinate across scopes to generate cross-functional integration tests.

Benefits:

- Dramatically improves speed of codebase comprehension.
- Enables scaling to very large codebases.
- Allows specialization among agents (e.g., test generation, fuzzing, documentation).

---

7. Non-Technical Stakeholder Interaction

Many product managers and stakeholders struggle to contribute meaningfully to technical work. This system empowers them to speak in natural language, trigger experiments, and receive explanations—without risking the integrity of the codebase.

- Natural language interfaces enable non-technical users (e.g., project managers, clients) to:
    - Propose feature ideas.

    - Request experiments in sandboxes.
    - Receive plain-language explanations of code functions.
- Proposed changes can lead to sandboxed experiments and safe pull request submissions for technical review.

Benefits:

- Bridges business-technical collaboration.
- Empowers product teams without risking stability.
- Accelerates feature ideation and innovation pipelines.

---

8. Private Key-Secured Operations

The system's sandbox can safely store and use private keys to sign commits or send blockchain transactions, without exposing keys to agents or external systems. This supports secure deployment and Web3-native workflows.

- Private cryptographic keys are stored securely inside sandboxes.
- Used to:
    - Sign commits or pull requests.
    - Sign blockchain transactions.
    - Establish secure encrypted communication channels.

Benefits:

- Enables blockchain interactions in tests.
- Prevents key leakage and unauthorized signing.
- Supports HIPAA, GDPR, PCI-DSS compliance.

9. Cryptographic Governance and Secure Upgrades

Upgrading the sandbox is tightly controlled via M-of-N cryptographic governance, enabling trusted upgrades while protecting against insider threats or unauthorized modifications. This mirrors enterprise security practices and supports regulated industries.

- Sandbox upgrades require quorum-signed approval by trusted auditors.
- Quorum rules (e.g., 2-of-3, 3-of-5) are enforced internally.
- Optional emergency access procedures require multi-party key custody authorization.

Benefits:

- Provides verifiable, audit-trail-controlled software evolution.
- Reduces insider risks.
- Supports enterprise-grade compliance demands.

Non-Obviousness of the Invention

The system described herein comprises a novel combination of components and workflows that, in their totality, would not have been obvious to a person having ordinary skill in the art (PHOSITA) at the time of invention. While some individual techniques—such as static analysis, code generation, or sandboxing—exist in isolation, the integration of bottom-up semantic comprehension, persistent structured memory, deterministic sandbox execution, agent swarms, cryptographic upgrade governance, and secure tool orchestration introduces a new architectural paradigm. No known prior art teaches or suggests this combination, nor would a PHOSITA have had a clear motivation to combine these features in the disclosed manner. The invention yields synergistic and unexpected benefits, including autonomous pull request generation, verified test synthesis, regulatory compliance by design, and reproducible agent reasoning — results not attainable by the prior state of the art.

Statement of Advantages and Unexpected Results

The disclosed system achieves a combination of outcomes that are not predictable from any individual component or known prior art. In particular:

- Bottom-up comprehension enables semantic understanding of codebases without relying on brittle top-down prompts or large-token context windows.

- Structured memory allows agent outputs to persist and accumulate, enabling reproducible reasoning and long-term stewardship of software projects.

- The harness enforces deterministic tool usage, contrasting with prompt-driven models like Copilot or ChatGPT that lack auditability or traceability.

- Secure sandboxing with M-of-N quorum-controlled upgrades provides a cryptographically enforceable boundary around agent behavior, supporting compliance with HIPAA, PCI-DSS, and other sensitive standards — functionality not present in known AI developer tools.

- Agent swarms allow horizontal parallelism and specialization across codebase dependency graphs, reducing the time to comprehension at scale while maintaining consistent knowledge integration.

- The use of natural language interfaces for non-technical stakeholders allows sandboxed experimentation and feature submission in a manner not seen in existing systems.

These advantages are **synergistic** rather than additive. For example, while static analysis tools may verify invariants and LLMs may generate text, no system combines automated comprehension, test generation, secure execution, and governance within a reproducible agent framework. The autonomous generation and validation of pull requests, documentation, and integration tests—without continuous human intervention—constitute an *unexpected and non-obvious capability* to a person having ordinary skill in the art at the time of the invention.

As a strict patent examiner, the following differentiators reinforce the non-obviousness of the disclosed system:

First, the system must be clearly distinguished from vector-based retrieval-augmented generation (RAG) platforms such as LangChain or Haystack. These frameworks rely on semantic similarity and embedding vectors to retrieve information, whereas the disclosed invention uses deterministic, path-based access via a `filePaths` structure, coupled with a statically precomputed `index.json` that maps hierarchical identifiers to precise line ranges in code files. This ensures traceability, reproducibility, and compatibility with version control diffs—none of which are guaranteed by vector-based RAG.

Second, the system includes specific implementations of how `index.json` is derived across language families. For example, Babel or Acorn may be used to parse JavaScript files and extract a tree of functions and objects with start/end positions; PHP code may be analyzed with `php-parser` to derive class/method structures; and config files (e.g., JSON, YAML) may be mapped directly to key paths. These indices enable token-efficient agent prompting and deterministic tool invocation.

Third, the invention departs significantly from commercial tools like Cursor, TabNine, and Amazon CodeWhisperer. While such tools may ingest full files or apply LLMs to IDE contexts, none expose or use a structured API that allows autonomous agents to request hierarchical paths into files and receive exact line-based hunks for comprehension and patching. The index-based context expansion, diff compatibility, and fallback mechanisms are not known to be implemented in those tools, either publicly or in patents.

Together, these architectural distinctions, operational mechanics, and user-facing behaviors contribute to a system that is not merely a novel arrangement of known components but a new paradigm for scalable, trustworthy AI-assisted software evolution.

Claims

1. A system for autonomous comprehension and maintenance of a software codebase, comprising:

- parsing the codebase into an intermediate representation;
- identifying code units without internal dependencies;
- analyzing said code units to extract semantic behavior;
- inductively combining the extracted semantics to understand dependent higher-level code units; and
- constructing a persistent structured memory representing the behavior of the codebase.

2. The system of claim 1, wherein for each code unit, the system:

- generates preconditions, postconditions, and invariants;
- generates one or more unit tests; and
- validates the orthogonality of each unit test by modifying the code to intentionally trigger isolated test failures.

3. The system of claim 2, wherein an agent proposes a correction for failed tests, reruns the updated test suite, and submits a pull request when the failure is resolved.

4. The system of claim 1, wherein each agent operates within a persistent sandboxed environment seeded deterministically and encrypting data at rest.

5. The system of claim 4, wherein sandbox environments receive software updates only via scheduled internal fetches and apply updates cryptographically signed by a quorum of auditors.

6. The system of claim 5, wherein emergency administrative access to the sandbox is gated through multi-party cryptographic key custody and quorum approval.

7. The system of claim 1, wherein agents communicate via a coordinator that executes structured tool instructions, including:

- storing or retrieving from structured memory;
- executing functions;

- issuing HTTP requests;
- committing or rolling back changes; or
- installing dependencies.

8. The system of claim 7, wherein the coordinator captures structured results from sandbox operations, including:

- compiler errors;
- runtime logs or warnings;
- HTTP status codes and payloads; and
- test results.

9. The system of claim 8, wherein the agent interprets structured results and determines the next action using a ReAct-style action-response loop.

10. The system of claim 1, wherein a swarm of autonomous agents operates in parallel, each assigned to a disjoint scope of the codebase.

11. The system of claim 10, wherein agents collaboratively generate integration tests to validate interactions between their respective scopes.

12. The system of claim 11, wherein integration tests include multi-user scenarios, time progression, or cross-module interactions replayed in sandbox environments.

13. The system of any of the foregoing claims, wherein one or more agents describe inferred workflows or behaviors in natural language for documentation or user interface integration.

14. The system of claim 1, wherein non-technical users interact with the system via a natural language interface to:

- propose feature ideas;
- initiate experiments; or
- receive documentation.

15. The system of claim 14, wherein proposed features are sandboxed, tested, and submitted to version control as structured pull requests.

16. The system of claim 4, wherein the sandbox environment contains a private key used to:

- cryptographically sign pull requests; or
- submit blockchain transactions without exposing the key externally.

17. The system of claim 16, wherein agents interact with forked blockchain ledgers or token balances in test environments to validate economic or cryptographic logic.

18. The system of claim 1, wherein all agent actions, tool calls, and received results are recorded in a structured log for reproducibility and auditability.

19. The system of claim 18, wherein agents may retrieve and reflect on their own past reasoning steps and results when planning future actions.

20. The system of claim 1, wherein sandbox environments maintain persistent state across sessions, allowing agents to install dependencies and build upon prior work without reinitialization.

21. The system of any of the foregoing claims, wherein agent instructions are scheduled via a task queue and executed deterministically, enabling reproducibility across replicas.

22. The system of any of the foregoing claims, wherein pull requests are submitted only after passing a quorum of tests, including unit, integration, and user acceptance tests generated by agents.

23. The system of claim 1, wherein structured memory includes data not only about code, but also documentation conventions, naming patterns, user stories, or past bug patterns.

Figure 1: Secure Sandboxed Agent Orchestration Architecture

Figure 1 illustrates the high-level system architecture for the disclosed invention. It represents a secure sandboxed environment where autonomous agents interact with codebases, databases, tools, and users to perform comprehension, testing, documentation, and maintenance tasks in a verifiable and reproducible manner.

This architecture comprises the following components:

1. **Sandbox Boundary**
    - All agent activities occur within a cryptographically secured, persistent sandbox environment that prevents any access to or mutation of production systems.
    - The sandbox includes forked infrastructure for databases and codebases, ensuring full reproducibility and safety.
2. **Structured Memory**
    - A long-term memory store that agents build and query.
    - Stores structured knowledge for each code module or function:
        - Unit and integration tests
        - Preconditions, postconditions, and invariants
        - Human-readable documentation
        - Natural language scenario descriptions (e.g., RAG-style memory entries, keywords)
    - Enables reproducible, auditable reasoning across sessions.
3. **Natural Language Interface and UX**
    - Provides a communication interface for both technical and non-technical users.
    - Users can:
        - Propose changes
        - Ask for code explanations
        - Trigger test runs or experiments in the sandbox
    - The system interprets user input and routes commands through the coordinator.
4. **Coordinator**
    - The orchestrator of all tool usage and agent task assignment.
    - Receives inputs from:
        - Natural language interface
        - Structured memory
        - Forked codebase and database
    - Interfaces with agents by providing:
        - Context
        - Tool invocation permissions
        - ReAct-style action-response loops
5. **Agent Swarm (CPU)**
    - A scalable pool of parallel, specialized agents that operate on isolated or cooperative scopes within the codebase.
    - Each agent may:
        - Analyze a function or class

        - Generate tests or documentation
        - Propose a patch or fix
        - Validate system behavior in forked environments
    - Operates deterministically to ensure reproducibility.
6. **Model API (GPU)**
    - Abstracted API used by agents to invoke large language models (LLMs).
    - Can operate with:
        - Local models for air-gapped deployments
        - Secure remote inference endpoints
    - Benefits include:
        - Security compliance (e.g., HIPAA, PCI)
        - Ability to encrypt data at rest
        - Custom model integration or fine-tuning
7. **Forked Codebase and Forked Database**
    - Cloned Git repositories and MySQL (or other DBMS) instances represent the working environment for agents.
    - Changes proposed by agents do not affect the original system.
    - Enables simulation, regression testing, and safe experimentation.
8. **Staging Database and Proposed Pull Requests**
    - Changes validated in the sandbox may be submitted to a staging database or proposed as pull requests.
    - Pull requests include:
        - Code changes
        - Generated tests
        - Documentation and reasoning trace
9. **User-Agent**
    - A user-facing monitoring or guidance interface.
    - Enables interaction with agent results and manual intervention when needed.

Interfaces and Benefits:

- **Interface Can Include:**
    - Full or partial context
    - Structured tool commands (e.g., `commit()`, `rollback()`)
    - ReAct-style reasoning with intermediate feedback loops
- **Benefits:**
    - All agent actions are deterministic and traceable.
    - Reproducible workflows enable compliance, debugging, and audit trails.
    - Enables natural language–driven workflows for non-technical users without sacrificing software safety.
    - Agent swarms ensure scalable, parallel comprehension of large codebases.
    - Secure model usage and sandboxed key access enable blockchain operations, cryptographic commits, and compliance automation.

This architecture, as shown in Figure 1, forms the operational backbone of the disclosed invention—allowing enterprise-grade AI-assisted software evolution with deterministic behavior, secure execution, and structured knowledge retention.

Figure 2: Swarm-Based Inductive Code Comprehension Pipeline

Figure 2 illustrates the workflow used by the agent swarm to perform bottom-up comprehension of a software codebase through a dependency-driven hierarchy. It depicts how multiple autonomous agents operate in parallel to understand individual code units and progressively reason about higher-order structures by combining verified results from lower-level dependencies.

Key elements in this architecture include:

1. **Swarm of Agents**
    - Agents (e.g., Agent A, Agent B) operate in parallel, each assigned to a disjoint or overlapping subset of functions or modules within the codebase.
    - The Coordinator assigns scopes and orchestrates execution and result integration.
2. **Hierarchical Function Levels**
    - Level 1 functions (`f1`, `g1`, `h1`) represent leaf functions—i.e., units with no internal dependencies.
    - Level 2 functions (`f2`, `g2`, `h2`) depend on one or more Level 1 functions and are analyzed only after their dependencies are verified.
    - The arrows indicate the bottom-up reasoning path, where verified behavior and test results from Level 1 inform the comprehension and testing of Level 2.
3. **Coordinator Role**
    - The Coordinator component:
        - Forks sandboxed environments
        - Assigns tasks to available agents
        - Maintains and queries structured memory
        - Orchestrates memory updates and test validation
        - Consolidates agent results
        - Submits validated pull requests to version control systems
4. **Execution Characteristics**
    - All analysis and reasoning occur deterministically in forked sandboxes.
    - Each agent constructs artifacts such as:
        - Unit and integration tests
        - Invariants and documentation
        - Contextual diffs
    - Memory and test results are stored in a structured, queryable form.

Benefits Illustrated by Figure 2:

- Promotes modular, bottom-up reasoning that avoids hallucination and scales to large codebases.
- Enables specialized agents to reason within their scope while remaining interoperable via shared memory.

- Allows massive parallelization and specialization (e.g., doc-writing agents, fuzzers, audit bots).
- Ensures that higher-order logic is derived from reproducibly validated primitives.
- Coordinates agent behavior in a transparent, traceable, and auditable manner.

Figure 2 exemplifies the core invention principle of inductive comprehension, where understanding is composed from verified units upward—unlike reactive, context-heavy top-down systems. This makes the approach particularly well-suited for enterprise-scale software environments with long-lived codebases and strong correctness requirements.

Figure 3: Path-Based Context Expansion and Index-Guided Code Extraction for Efficient Agent Reasoning

Figure 3 expands on the secure sandbox environment shown in Figure 1 by illustrating how agents dynamically and efficiently acquire only the relevant slices of context needed to reason about code, without ingesting entire files. This is accomplished via a structured path-based mechanism and static indexing.

Key Components:

1. **Structured Memory**
    - Continuously built and queried by agents
    - Stores unit tests, integration tests, invariants, documentation, and natural language summaries for each function or scenario
    - Allows reproducible memory reads across sessions
2. **Natural Language Interface and UX**
    - Allows users—especially non-technical ones—to trigger experiments, propose features, or inspect explanations
    - Feeds requests to the Coordinator, which converts them to structured agent tasks
3. **Coordinator**
    - Orchestrates all agent execution and manages:
        - Memory access
        - File and database forking
        - Model invocation
        - Diff generation
        - Task scheduling
    - Can include structured instructions: `context`, `tool use`, `ReAct pattern`
4. **Agent Swarm (CPU)**
    - Performs comprehension and maintenance tasks
    - Dynamically issues `filePaths` requests for specific portions of code or data
    - Operates in coordination using structured memory and path-based reasoning
5. **Model API (GPU)**
    - Invoked as needed for language understanding
    - Can be local or remote, while keeping data encrypted and within compliance boundaries
6. **Forked Database and Forked Codebase**
    - Sandboxed versions of production-like systems for safe execution
    - Allow agents to run tests and generate changes
    - Forked repos are used to propose and track structured pull requests
7. **Staging Database and User-Agent Interface**
    - Accepts validated changes and user feedback
    - Users can review or approve pull requests generated by the system

Path-Based Context Expansion and Indexing (Core Mechanism):

Figure 3 illustrates a system for dynamic context expansion, allowing autonomous agents to request, retrieve, and reason about targeted parts of a codebase or configuration files without requiring full-file ingestion.

This mechanism includes the following elements:

1. **FilePath-Based Requesting**
   - Agents can request specific portions of a file using hierarchical paths (e.g., `["features", "telemetry"]` for JSON/YAML or `["MyClass", "myMethod"]` for code).
   - These requests are declared in the `filePaths` object, with each filename mapped to an array of path arrays.
2. **Structured Index for Source Code Files**
   - For JavaScript, PHP, and other code files, a statically precomputed `index.json` maps path arrays to line number ranges (`fromLine`, `toLine`), types (`class`, `method`, `object`, etc.), and optionally scope types.
   - This index is generated using language-specific parsers (e.g., `babel`, `esprima`, `php-parser`) and enables efficient slicing of file hunks for agent prompts.
3. **Adapter-Based Extraction**
   - A language-agnostic adapter engine receives the file path and path arrays.
   - For structured config files (e.g., JSON, YAML, TOML), the adapter walks the hierarchical structure to extract exact matching blocks.
   - For source code files, it uses the `index.json` to locate and extract the exact line ranges corresponding to each path.
   - Line numbers are preserved for inclusion in diff outputs.
4. **Agent-Initiated Context Expansion Protocol**
   - Agents may initially propose small diffs based on limited context.
   - If additional information is required, agents include a `request` object in `results.json`, listing:
     - Specific `filePaths` (path arrays)
     - Or a full `file` fallback if the scope is unclear
     - Or an `index` request to receive the file's full `index.json` subtree
5. **Fallback Modes and Heuristics**
   - If the requested path is ambiguous or the model is unsure, it may fall back to requesting the whole file.
   - The harness may decide to send entire files based on:
     - Small file size
     - Absence of a usable index
     - Previous errors from failed diffs
6. **Harness-Driven Indexing and Querying**
   - The harness interprets structured requests and invokes relevant adapters to return:

        - Minimal file hunks
        - Indexed summaries (for `index` requests)
        - Complete file context (for `file` requests)
    - This system enables memory-efficient, deterministic expansion of context.

Benefits Illustrated by Figure 3:

- Supports context-on-demand behavior for autonomous agents.
- Enables fine-grained access to source and structured config files.
- Minimizes token use and bandwidth in prompt contexts.
- Supports secure and deterministic behavior in accordance with the sandbox policies.
- Lays groundwork for future agent specialization, such as auto-completion, transformation, or documentation generation based on scoped information.
- Enables efficient token use for LLMs by limiting context to exact file hunks
- Makes context access deterministic, reproducible, and diff-compatible
- Keeps agent workflows secure and interpretable across distributed teams
- Empowers agents to scale codebase comprehension while retaining auditability
- Seamlessly integrates with Figures 1 and 2 by feeding context into the same agent-coordinator architecture

This context-expansion protocol and adapter-index architecture bridges the gap between large codebase scale and the token/latency constraints of LLM-driven agents, enabling secure, efficient, and comprehensible evolution of modern software systems.

Figure 4: Expert Groker Delegation and KV-Primed Evolutionary Agent Specialization

Figure 4 builds upon the agent infrastructure described in previous figures by introducing a powerful strategy for delegating subtasks to specialized expert Grokers, each of which is prewarmed with domain-specific context via KV cache priming. This architecture enables efficient, interpretable, and modular task execution across large-scale systems—while also supporting dynamic optimization through evolutionary selection of high-performing Grokers.

Key Components:

Primary Groker (Task Orchestrator)
The Primary Groker operates at a higher level of abstraction, accepting tasks from users or other systems and decomposing them into functional subtasks. It determines which expert Grokers to invoke based on semantic fit, performance history, and architectural structure (e.g., matching to a file path, class, or component type).

Expert Grokers with Cached KV State
Each expert Groker is a specialized agent with:

- A persistent prefix context describing its domain (e.g., Auth, SQL, UI)
- A cached KV tensor state for fast generation without recomputing common instructions
- Embedded behavioral traits, style guides, and testing conventions relevant to its role

This structure enables low-latency, high-consistency outputs, especially in environments where the same operations (e.g., validating JWTs, formatting SQL joins) recur across requests.

Function-Specific Delegation
Subtasks are routed to Grokers based on capability:

- `Codegen()` – For controller and handler generation
- `QueryPlan()` – For SQL query construction and optimization
- `RenderForm()` – For UI layout and interactive component wiring

Each Groker can return results, diffs, or even request additional context expansions.

Merged Output Composer

The Primary Groker integrates the outputs into a complete result—such as a pull request, patch, or deployable module—ensuring coherence, compatibility, and staging readiness.

Evolutionary Groker Selection and Genetic Optimization

A key innovation in this system is its ability to evolve Groker agents over time, using concepts borrowed from genetic algorithms:

Forking Grokers
A Groker can be forked by:

- Copying its prefix instructions and modifying them slightly
- Mutating prompts, prompt weights, or behavior heuristics
- Replacing or augmenting parts of its cached context (e.g., swap in new examples, modify output constraints)

Each forked Groker variant inherits the core skills of its ancestor while exploring a new variation in behavior or specialization.

Evaluating Fitness
Forked Grokers are run on the same or similar tasks and scored based on:

- Correctness or acceptance of output (e.g., test pass rate)
- Performance (e.g., token count, speed)
- Alignment with style guides, security constraints, or user feedback

Selecting Survivors
The system maintains a memory of top-performing Grokers, which are:

- Retained for future reuse
- Used as new "parents" for further mutations
- Cached in a registry or memory pool with model affinity

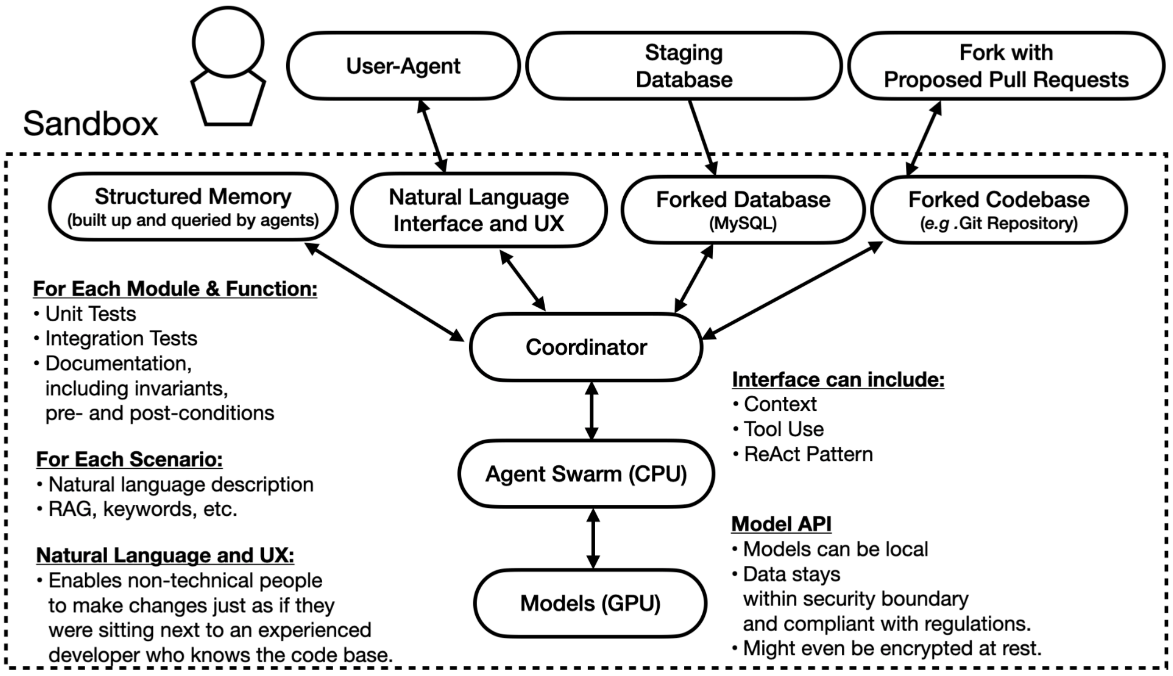
KV-Aware Evolution
Since each Groker maintains its own KV cache, forks can reuse and slightly adjust that cached

tensor state to accelerate downstream runs. This allows for fast mutation-exploration cycles without recomputing context-heavy instructions.

Benefits Illustrated by Figure 4:

- Modular delegation of subtasks improves clarity and reduces token bloat.
- KV cache reuse enables low-latency generation for domain-specific logic.
- Semantic specialization improves alignment with architectural conventions.
- Evolutionary strategies allow Grokers to optimize over time, converging on best practices through empirical feedback.
- Forking and selection create a population of agents tuned to specific codebases or teams.
- Auditable agent trees preserve lineage and mutation history for compliance and debugging.
- Cross-agent memory sharing allows for intelligent knowledge transfer without polluting shared context windows.

This architecture allows for multi-agent co-evolution within a secure, orchestrated runtime, combining structured memory access, dynamic delegation, and evolutionary pressure to continuously improve system performance while preserving explainability and fine-grained control.


Sandbox
User-Agent
Staging Database
Fork with Proposed Pull Requests
Structured Memory (built up and queried by agents)
Natural Language Interface and UX
Forked Database (MySQL)
Forked Codebase (*e.g* .Git Repository)
Coordinator
Agent Swarm (CPU)
Models (GPU)

For Each Module & Function:
- Unit Tests
- Integration Tests
- Documentation, including invariants, pre- and post-conditions

For Each Scenario:
- Natural language description
- RAG, keywords, etc.

Natural Language and UX:
- Enables non-technical people to make changes just as if they were sitting next to an experienced developer who knows the code base.

Interface can include:
- Context
- Tool Use
- ReAct Pattern

Model API
- Models can be local
- Data stays within security boundary and compliant with regulations.
- Might even be encrypted at rest.

Figure 1

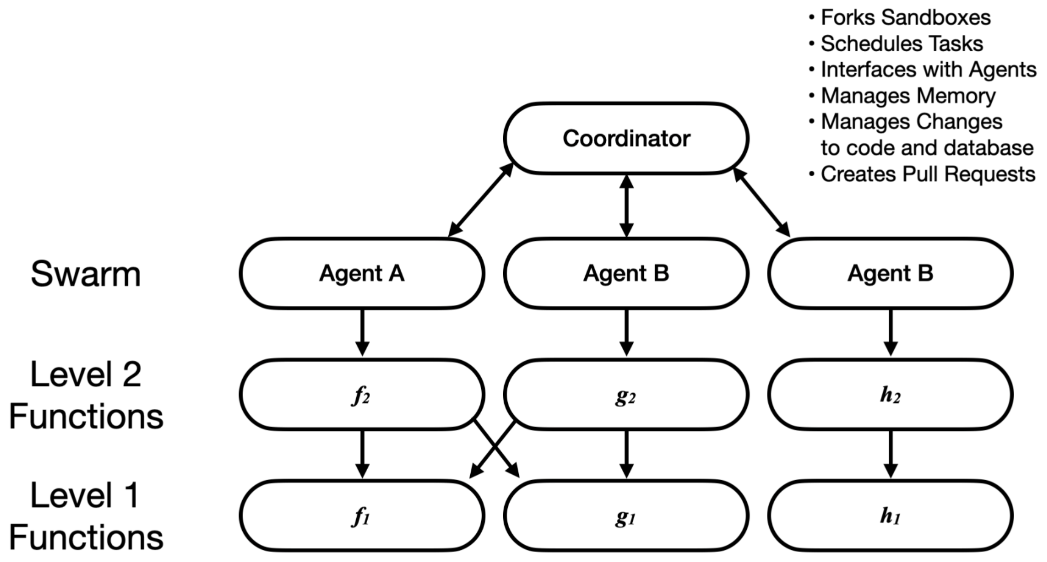
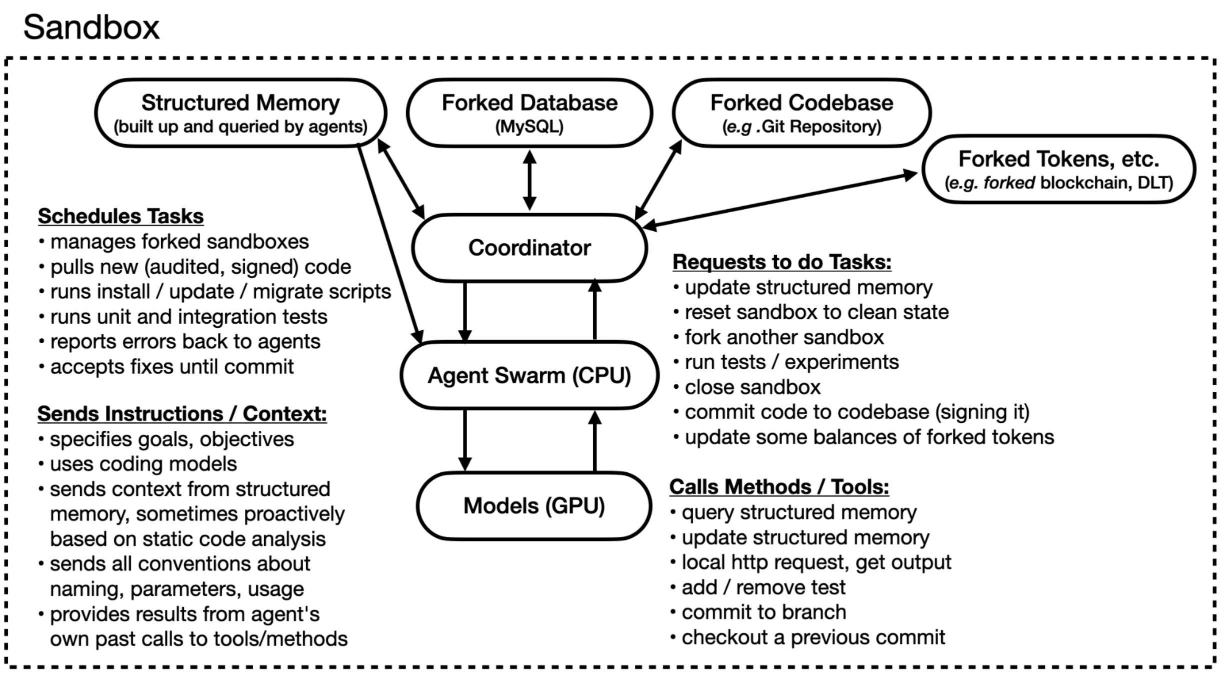

• Forks Sandboxes
• Schedules Tasks
• Interfaces with Agents
• Manages Memory
• Manages Changes to code and database
• Creates Pull Requests
Coordinator
Swarm
Agent A
Agent B
Agent B
Level 2 Functions
f_2
g_2
h_2
Level 1 Functions
f_1
g_1
h_1

Figure 2



Schedules Tasks
- manages forked sandboxes
- pulls new (audited, signed) code
- runs install / update / migrate scripts
- runs unit and integration tests
- reports errors back to agents
- accepts fixes until commit

Sends Instructions / Context:
- specifies goals, objectives
- uses coding models
- sends context from structured memory, sometimes proactively based on static code analysis
- sends all conventions about naming, parameters, usage
- provides results from agent's own past calls to tools/methods

Requests to do Tasks:
- update structured memory
- reset sandbox to clean state
- fork another sandbox
- run tests / experiments
- close sandbox
- commit code to codebase (signing it)
- update some balances of forked tokens

Calls Methods / Tools:
- query structured memory
- update structured memory
- local http request, get output
- add / remove test
- commit to branch
- checkout a previous commit

Figure 3

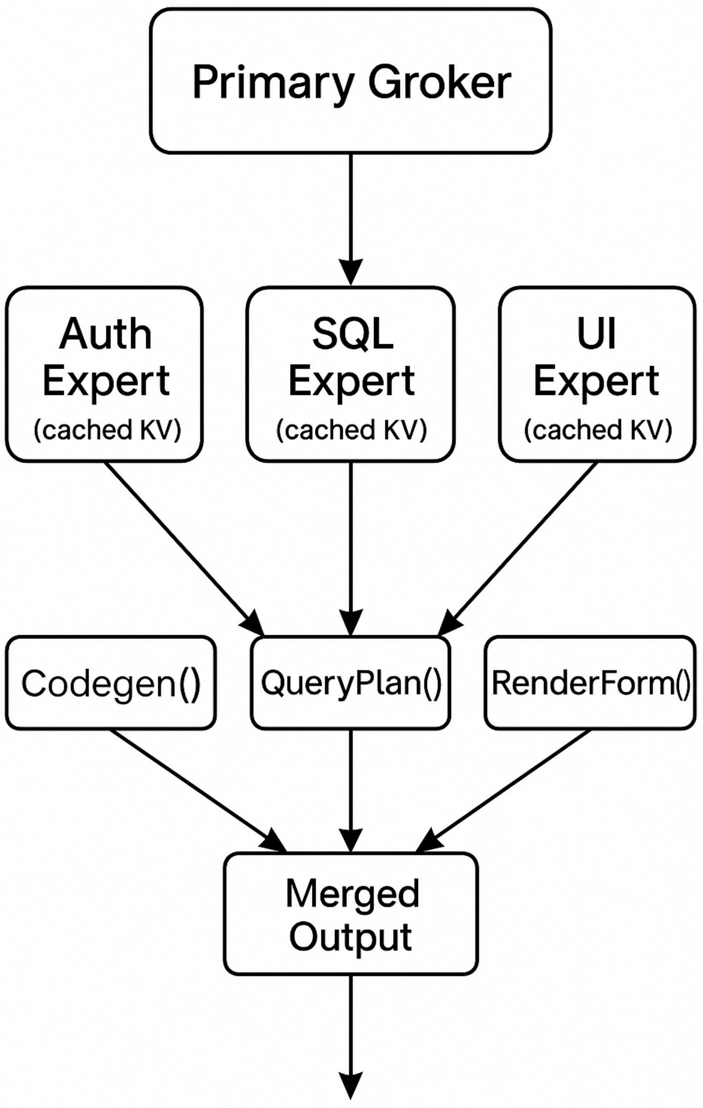

Primary Groker
Auth Expert (cached KV)
SQL Expert (cached KV)
UI Expert (cached KV)
Codegen()
QueryPlan()
RenderForm()
Merged Output

Figure 4