# Product Requirements Document: Autonomous Self-Evolving Software Engineer (AI SWE)

## 1. Executive Summary

### 1.1 Product Vision and Strategic Necessity

The global software engineering landscape is currently navigating a profound and irreversible paradigm shift. We are moving from the era of "human-centric creation augmented by tools" (the Copilot era) to a new epoch of "agent-centric generation supervised by humans" (the Agentic era). This Product Requirements Document (PRD) articulates the comprehensive design and functional specifications for the **Autonomous Self-Evolving Software Engineer (AI SWE)**, an advanced Agentic AI system designed to function as a resilient, closed-loop operational entity capable of active reasoning, long-term planning, and autonomous execution.1

Current generative AI tools, such as GitHub Copilot or ChatGPT, operate primarily as reactive, stateless assistants. They require continuous human prompting to maintain context and drive progress. While they accelerate individual coding tasks, they do not fundamentally alter the mechanics of software delivery; the human remains the bottleneck for context switching, verification, and integration. The AI SWE acts as a categorical leap beyond this technology. It is architected to function effectively as a junior-to-mid-level software engineer, capable of ingesting high-level natural language requirements, decomposing them into technical specifications, implementing production-grade code, executing rigorous testing cycles, managing deployment, and—crucially—evolving its own capabilities through **Reinforcement Learning from Execution Feedback (RLEF)**.1

This initiative addresses the urgent "scalability bottleneck" facing modern engineering organizations. As systems grow in complexity (microservices, distributed architectures), the cognitive load on human engineers becomes unsustainable. Industry analysis by Gartner forecasts that "Agentic AI" will govern over 40% of enterprise applications by 2026. The AI SWE is not merely a productivity tool; it is a strategic asset designed to compress the Software Development Life Cycle (SDLC) by automating the iterative "inner loop" of coding and testing, potentially unlocking a 20-60% increase in velocity for routine engineering tasks.1

### 1.2 Core Value Proposition

The AI SWE delivers value across three primary dimensions:

1. **Velocity and Throughput Amplification:** By offloading routine implementation, testing, and migration tasks to autonomous agents, human engineers are freed to focus on high-value architectural design and innovation. Agents operate asynchronously and in parallel, enabling a "Follow-the-Sun" engineering model where features are developed, tested, and prepped for review 24/7.1
2. **Reduction of Technical Debt:** Human engineers, under pressure to meet deadlines, often compromise on code quality, leading to accumulated technical debt. The AI SWE, constrained by strict architectural governance and unburdened by fatigue, can enforce coding standards and perform background refactoring continuously, ensuring long-term codebase health.1
3. **Operational Resilience via Self-Evolution:** The defining technical differentiator of this system is its ability to learn from failure. Unlike static models, the AI SWE utilizes an RLEF loop to ground its logic in the "ground truth" of compiler outputs and test results. When it encounters a runtime error, it does not stop; it analyzes the stack trace, hypothesizes a fix, and applies remedial patches autonomously, mimicking the experiential learning curve of a human developer.1

### 1.3 Strategic Alignment with Enterprise Goals

The successful deployment of the AI SWE aligns with broader enterprise goals of operational efficiency and cost control. By leveraging a **Local-First** technology stack (utilizing the open-weight **Qwen2.5-Coder 7B** model and **Ollama** inference engine), the system ensures strict data privacy and eliminates the risk of Intellectual Property (IP) leakage associated with public cloud API calls. Furthermore, the use of efficient, smaller-parameter models drastically reduces the marginal cost of code generation compared to reliance on frontier models like GPT-4, making high-volume autonomous reasoning economically viable.3

## 2. Problem Statement and User Personas

### 2.1 The "Scalability Bottleneck" in Modern Engineering

Modern software engineering is plagued by a fundamental inefficiency: the disconnect between "intent" and "implementation." A senior architect can often define the solution to a problem in minutes ("We need a Redis cache here to reduce DB load"), but the actual implementation—writing the boilerplate, configuring the client, writing unit tests, updating the Dockerfile, and verifying the deployment—can take hours or days. This latency is exacerbated by context switching, meeting interruptions, and the cognitive fatigue of managing complex dependency graphs.

Furthermore, the "Junior Engineer Paradox" presents a challenge: while organizations need to hire junior engineers to scale, the time senior engineers spend mentoring and reviewing junior code often negates the productivity gains in the short term. The AI SWE addresses this by acting as a "Digital Junior Engineer" that requires supervision but does not consume human mentorship time for repetitive instruction, and which improves its performance without needing emotional support or career management.1

### 2.2 Target User Personas

To ensure the system meets the needs of its human supervisors, we define three primary user personas.

#### 2.2.1 Persona A: The Engineering Manager (The Supervisor)

* **Role:** Oversees team velocity, budget, and project timelines.
* **Pain Points:** Lack of visibility into bottlenecks; high cost of simple feature delivery; "bus factor" risks where knowledge is siloed in specific humans.
* **Needs:** A dashboard overview of agent activity; clear metrics on "Cost per Feature"; ability to assign backlog tickets to the AI agent during off-hours.
* **Interaction Mode:** Web Dashboard, Weekly Reports.
* **Key User Story:** "As an Engineering Manager, I want to assign a backlog of low-priority bug fixes to the AI SWE so that my human team can focus on the critical Q4 deliverables.".6

#### 2.2.2 Persona B: The Senior Software Architect (The Specifier)

* **Role:** Defines system architecture, selects technologies, and ensures code quality.
* **Pain Points:** Spending too much time reviewing boilerplate code; explaining the same design patterns repeatedly; cleaning up "spaghetti code."
* **Needs:** A system that strictly follows defined architectural patterns (e.g., "Always use the Repository pattern"); a tool that generates the scaffolding for complex systems automatically.
* **Interaction Mode:** Markdown PRDs, Code Review Interface.
* **Key User Story:** "As an Architect, I want to upload a high-level architecture diagram and have the AI SWE generate the corresponding project structure, Dockerfiles, and CI/CD pipelines compliant with our security standards.".8

#### 2.2.3 Persona C: The Developer (The Collaborator)

* **Role:** Writes code, fixes bugs, and maintains the application.
* **Pain Points:** Writing unit tests; updating documentation; upgrading dependencies; debugging obscure environment errors.
* **Needs:** A "force multiplier" that can handle the tedious parts of the job (e.g., "Write tests for this module") while they focus on business logic.
* **Interaction Mode:** IDE Extension, CLI, Pull Request comments.
* **Key User Story:** "As a Developer, I want the AI SWE to analyze my failed build logs, identify the root cause of the dependency conflict, and open a Pull Request with the fix.".7

### 2.3 Detailed User Stories and Acceptance Criteria

| **ID** | **Persona** | **User Story** | **Acceptance Criteria** |
| --- | --- | --- | --- |
| **US-01** | Architect | Ingest Natural Language PRD | System parses a text/markdown PRD.  System identifies all functional requirements.  System produces a visualized Task Graph (DAG) for approval. |
| **US-02** | Developer | Generate Unit Tests | System generates Pytest files for a given module.  Tests cover >80% branch coverage.  Tests pass in the Docker sandbox. |
| **US-03** | Manager | Autonomous Bug Fix | System detects a runtime error in staging.  System analyzes stack trace and commits a fix.  System verifies fix with regression tests.  No human intervention required. |
| **US-04** | Security | Vulnerability Scan | System scans all generated code with Bandit/Semgrep.  System automatically rejects code with High/Critical CVEs.  System provides remediation suggestions. |
| **US-05** | Developer | Code Explanation | System provides a "Thought Log" explaining why specific libraries were chosen.  System links code decisions to specific PRD requirements. |

## 3. Product Objectives and Scope

### 3.1 Strategic Objectives

1. **Validate Autonomy:** Prove that an agentic system can traverse the full "Idea-to-Code" loop without human hand-holding for tasks of moderate complexity (e.g., "Build a CRUD API with Auth").
2. **Demonstrate Self-Healing:** Establish a robust **RLEF** loop where the system uses compiler/test feedback to iteratively repair its own code, achieving a "Self-Correction Rate" of >60%.1
3. **Ensure Economic Viability:** Prove that high-volume reasoning is cost-effective by utilizing the **Qwen2.5-Coder 7B** model, demonstrating a cost-per-resolution significantly lower than human rates.4
4. **Secure by Design:** Implement a "Defense-in-Depth" strategy using **Docker** sandboxing, non-human identity management, and static analysis gates to mitigate the risks of rogue agent behavior (the "Moltbook" risk).10

### 3.2 Scope of Capabilities

#### 3.2.1 In-Scope Capabilities (MVP)

The MVP will focus on "Greenfield" development of web services to minimize context-loading complexity.

* **Requirement Analysis:** Parsing natural language text into structured JSON task lists.
* **Polyglot Implementation:** Supporting **Python** (FastAPI, Django) and **JavaScript/TypeScript** (React, Node.js) as the primary ecosystems.1
* **Test-Driven Development:** The agent must generate tests *before* or *alongside* implementation. No code is committed without passing tests.
* **Infrastructure Management:** Generation of Dockerfile, docker-compose.yml, and Kubernetes manifests.
* **Iterative Repair:** A maximum of 5 autonomous retry attempts per task upon failure.12
* **Memory Persistence:** Utilization of **ChromaDB** for semantic context and **SQLite** for episodic history.

#### 3.2.2 Out-of-Scope (Phase 1)

* **Legacy Code Refactoring:** Navigating large, undocumented monolithic codebases ("Brownfield" projects) is excluded due to context window limitations.
* **Creative UI Design:** The agent will use standard component libraries (e.g., Material UI) but will not perform subjective visual design tasks.
* **Production Deployment:** The agent will deploy to a *sandboxed* Kubernetes cluster, not live production environments, to ensure safety during the PoC phase.
* **Voice/Video Processing:** Multimodal inputs are excluded; text/code is the only interface.

## 4. Proposed Solution Architecture

The architecture of the AI SWE is a departure from the "Chatbot" paradigm. It adopts a **Multi-Agent System (MAS)** architecture, where specialized agents collaborate under the supervision of a central orchestration state machine. This design mimics the structure of a human engineering team, separating concerns between planning, execution, verification, and review.

### 4.1 High-Level Architecture Diagram

The system is stratified into five distinct layers:

1. **Interaction Layer:** The interface for user intent (CLI/Web Dashboard).
2. **Orchestration Layer (Control Plane):** A stateful graph engine (**LangGraph**) managing the workflow.
3. **Agent Layer:** Specialized personas driven by the **Qwen2.5-Coder** LLM.
4. **Memory Layer:** Persistent storage for context (**ChromaDB**) and history (**SQLite**).
5. **Execution Layer:** The **Docker** sandbox where code meets reality.

### 4.2 Detailed Technology Stack Selection

The following technology stack has been selected to meet the requirements of performance, privacy, and modularity.1

| **Component** | **Technology** | **Justification** |
| --- | --- | --- |
| **Foundation Model** | **Qwen2.5-Coder 7B** | Selected for its SOTA performance on coding benchmarks (HumanEval, MBPP, SWE-bench), outperforming larger models like GPT-4 in specific coding tasks while being lightweight enough for local inference.13 |
| **Inference Engine** | **Ollama 0.1.29** | Provides a production-grade local inference server compatible with OpenAI APIs. Ensures data privacy (no code leaves the premises) and low-latency token generation.1 |
| **Orchestrator** | **LangGraph 0.0.58** | Unlike linear chains (LangChain), LangGraph enables *cyclical* state graphs. This is essential for the "Test-Fail-Fix-Test" loop, allowing the agent to revisit previous states based on feedback.1 |
| **Semantic Memory** | **ChromaDB 0.4.22** | Open-source vector database optimized for AI. Stores embeddings of the codebase, documentation, and architectural patterns for Retrieval-Augmented Generation (RAG).1 |
| **Episodic Memory** | **SQLite + SQLModel** | Lightweight, relational database to store the structured "Event Log" of agent actions. SQLModel provides Pythonic ORM interaction.1 |
| **Execution Sandbox** | **Docker Engine 25.x** | Provides hard isolation for code execution. Every task runs in an ephemeral container to prevent file system damage or malicious networking.1 |
| **Testing Framework** | **Pytest** | The industry standard for Python testing. Its rich plugin ecosystem and structured output make it ideal for parsing by the AI agent.1 |
| **Security Scanning** | **Bandit & Semgrep** | Deterministic static analysis tools. Bandit checks for Python security issues; Semgrep enforces custom policy rules.1 |

### 4.3 The Orchestration Engine: LangGraph

The heart of the AI SWE is the **LangGraph Control Plane**. Unlike simple "Chain of Thought" scripts, LangGraph manages a persistent StateGraph that represents the entire lifecycle of a coding task.

**State Schema:**

The global state object passed between agents contains the following fields:

Python

class AgentState(TypedDict):  
 project\_id: str # Unique identifier for the session  
 requirements: str # The original user prompt  
 plan: List # The DAG of decomposed tasks  
 current\_task: Task # The active task being worked on  
 code\_context: Dict[str, str] # Map of file paths to content  
 test\_results: TestResult # Outcome of the latest test run  
 error\_logs: List[str] # Stack traces from recent failures  
 iteration\_count: int # Counter to prevent infinite loops  
 memory\_retrieval: List[str] # Context fetched from ChromaDB

**Workflow Transitions:**

1. **Plan:** The **Planner Agent** populates the plan field.
2. **Execute:** The **Implementation Agent** reads current\_task, writes code, and updates code\_context.
3. **Verify:** The **Test Agent** runs the code in Docker and updates test\_results.
4. **Decide (Conditional Edge):**
   * **Success:** If test\_results.passed is True, transition to **Reviewer Agent**.
   * **Failure:** If test\_results.passed is False, increment iteration\_count.
     + If iteration\_count < MAX\_RETRIES: Transition back to **Implementation Agent** with error\_logs as input.
     + If iteration\_count >= MAX\_RETRIES: Transition to **Human Escalation**.
5. **Review:** The **Reviewer Agent** runs Bandit/Semgrep. If pass, commit. If fail, loop back.

### 4.4 The "Brain": Qwen2.5-Coder 7B

The choice of **Qwen2.5-Coder 7B** is strategic. Trained on 5.5 trillion tokens, it specializes in code generation, reasoning, and fixing.14

* **Performance:** It rivals GPT-4o in coding benchmarks like HumanEval and MBPP while being a fraction of the size.
* **Context Window:** It supports up to 128k tokens (though we will limit to 32k for speed in the PoC), allowing it to ingest significant portions of documentation and codebase context without "forgetting".14
* **Cost:** Running locally via Ollama means zero marginal cost per token, enabling the system to "think" (generate thousands of tokens of reasoning) without incurring API bills. This enables **"System 2" thinking**—spending compute time to verify and plan deeply.4

## 5. The Agent Workforce: Persona Specifications

The AI SWE is not a single model but a system of specialized agents, each initialized with a specific system prompt and toolset.

### 5.1 The Planner Agent (The Architect)

* **Objective:** Translate ambiguity into structure.
* **System Prompt:** "You are a Senior Software Architect. Your goal is to decompose a high-level requirement into a set of atomic, interdependent engineering tasks. You must identify circular dependencies and logical execution order. Output a Directed Acyclic Graph (DAG) in JSON format."
* **Tools:**
  + search\_docs(query): Query ChromaDB for architectural patterns (e.g., "FastAPI best practices").
  + read\_file(path): Inspect existing codebase to ensure consistency.
* **Output:** A plan.json file defining the roadmap.

### 5.2 The Implementation Agent (The Coder)

* **Objective:** Execute a single, bounded task from the plan.
* **System Prompt:** "You are an expert Polyglot Developer specializing in Python and TypeScript. You write clean, typed, and documented code adhering to SOLID principles. You have direct access to the file system. Do not use placeholders; write full implementation."
* **Tools:**
  + write\_file(path, content): Create or update source files.
  + list\_dir(path): Explore the directory structure.
  + run\_syntax\_check(path): Verify Python syntax before saving.
* **Constraint:** The agent works in a "Thought-Action-Observation" loop, reasoning about the code before writing it.

### 5.3 The Test Agent (The QA Engineer)

* **Objective:** Adversarial verification.
* **System Prompt:** "You are a QA Automation Engineer. Your goal is to break the code written by the Implementation Agent. Generate comprehensive unit and integration tests using Pytest. Cover edge cases, null inputs, and boundary conditions."
* **Tools:**
  + run\_test(path): Execute Pytest in the Docker container.
  + parse\_coverage(path): specific tool to check code coverage %.
* **Behavior:** The Test Agent operates in an adversarial loop. It is rewarded for finding bugs, incentivizing it to be thorough.

### 5.4 The Reviewer Agent (The Gatekeeper)

* **Objective:** Security and Code Quality assurance.
* **System Prompt:** "You are a Security Auditor and Senior Code Reviewer. Analyze the proposed changes for security vulnerabilities (OWASP Top 10), performance bottlenecks, and style violations."
* **Tools:**
  + run\_bandit(path): Static Application Security Testing (SAST).
  + run\_semgrep(path): Policy enforcement.
  + check\_style(path): Linting (Black/Prettier).
* **Policy:** "Deny by Default." Any finding with Severity > Medium blocks the PR.

### 5.5 The Runtime Agent (The SRE)

* **Objective:** Post-deployment monitoring and self-healing.
* **System Prompt:** "You are a Site Reliability Engineer (SRE). Monitor the application logs and metrics. If a crash or latency spike occurs, correlate it with recent changes and trigger a rollback or fix."
* **Tools:**
  + read\_logs(container\_id): Access runtime logs.
  + restart\_service(service\_name): Infrastructure control.
  + query\_metrics(): Fetch CPU/RAM usage.

## 6. Functional Requirements

### 6.1 Requirement Analysis & Task Decomposition

* **FR-01:** The system shall accept input in the form of raw text, Markdown files, or Jira ticket URLs.
* **FR-02:** The **Planner Agent** shall decompose requests into tasks with a maximum estimated duration of 1 hour (agent time) to ensure manageability.
* **FR-03:** The system shall identify implicit requirements. For example, a request for "User Login" must automatically trigger tasks for "Password Hashing," "Session Management," and "Secure Cookie Storage."

### 6.2 Autonomous Code Generation & Polyglot Support

* **FR-04:** The system shall generate code compatible with **Python 3.11.7** and **Node.js 20.x**.
* **FR-05:** Dependency management must be autonomous. The agent shall detect missing imports and update requirements.txt or package.json automatically.
* **FR-06:** The system shall generate fully functional **Dockerfiles** following best practices (multi-stage builds, non-root users) to ensure the application is deployable.

### 6.3 The RLEF Loop (Self-Correction Mechanism)

* **FR-07 (Feedback Integration):** Upon test failure, the system must feed the full stack trace and error message back to the Implementation Agent's context window.
* **FR-08 (Reasoning Requirement):** The agent must output a "Diagnosis" section before attempting a fix, explaining *why* the error occurred (e.g., "I attempted to access key 'x' but the API returns key 'y'").
* **FR-09 (Retry Logic):** The system shall support a configurable MAX\_RETRIES (default: 5). If the agent fails to fix the bug after 5 attempts, the task is marked as "Failed" and escalated to a human.
* **FR-10 (Success Reward):** Successful fixes (where tests pass) shall be logged to **Episodic Memory** (SQLite) to reinforce the strategy used.

### 6.4 Deployment & Infrastructure as Code (IaC)

* **FR-11:** The system shall utilize **Docker Compose** to spin up dependent services (e.g., a Postgres DB) required for integration testing.
* **FR-12:** The system shall verify the health of deployed containers using docker ps and HTTP health checks (curl localhost:8000/health).

## 7. Non-Functional Requirements

### 7.1 Performance and Scalability

* **NFR-01 (Inference Latency):** The local **Ollama** instance must serve tokens at a rate of at least **30 tokens/second** on the target hardware (consumer GPU with 24GB VRAM, e.g., RTX 3090/4090) to ensure the feedback loop is tight.19
* **NFR-02 (Parallelism):** The Orchestrator shall support parallel execution of non-dependent tasks. For example, "Write Frontend Component A" and "Write Backend API B" should proceed simultaneously if the Planner marks them as independent.
* **NFR-03 (Context Management):** The system must implement "Context Compaction." As the conversation grows, older turns must be summarized to keep the active context within the 32k token efficient window of Qwen2.5, preventing "Lost-in-the-Middle" performance degradation.

### 7.2 Security and Compliance

* **NFR-04 (Sandboxing):** All agent-generated code must be executed within ephemeral **Docker** containers. These containers must have **no network access** to the host or the public internet, except for a whitelisted proxy for installing packages (PyPI/npm).1
* **NFR-05 (Semantic Firewall):** An intermediate "Guardrail Model" (a lightweight classifier) must inspect all agent outputs *before* they are executed. It scans for "Jailbreak" attempts, exfiltration of system prompts, or malicious shell commands (e.g., rm -rf /).10
* **NFR-06 (Secret Hygiene):** The system shall strictly enforce the use of environment variables for secrets. It must never hardcode API keys or passwords in the source code. The **Reviewer Agent** is responsible for enforcing this via regex and entropy checks.21

### 7.3 Observability and Trust

* **NFR-07 (The "Glass Box" Requirement):** The system must provide a real-time "Thought Log" visualization on the dashboard. This log displays the agent's internal monologue ("I am checking the database schema...", "Tests failed, analyzing stack trace...") to build user trust and aid debugging.22
* **NFR-08 (Audit Trail):** Every action taken by the system (file edits, shell commands) must be logged to the **SQLite** database with a timestamp, agent ID, and the specific prompt that triggered it. This creates an immutable audit trail for forensic analysis.24

## 8. Data Strategy and Memory Architecture

A robust memory system is what separates an "Agent" from a "Chatbot." The AI SWE utilizes a dual-memory architecture.1

### 8.1 Episodic Memory (The "Black Box" Recorder)

Implemented using **SQLite** and **SQLModel**, this component records the chronological history of the agent's life.

* **Schema:**  
  Python  
  class AgentEvent(SQLModel, table=True):  
   id: Optional[int] = Field(default=None, primary\_key=True)  
   session\_id: str  
   timestamp: datetime  
   agent\_role: str  
   action: str # e.g., "EditFile", "RunTest"  
   input\_context: str # The prompt provided to the agent  
   output\_content: str # The agent's response  
   outcome: str # "Success", "Failure", "Error"  
   error\_signature: str # Hash of the stack trace (if any)
* **Utility:** This log is queried to prevent repetitive mistakes. Before running a command, the agent checks: "Have I run this command before in this session and failed?"

### 8.2 Semantic Memory (The Knowledge Base)

Implemented using **ChromaDB**, this component acts as the agent's reference library.

* **Content:**
  1. **Codebase Embeddings:** Every file in the project is chunked and embedded. This allows the agent to answer "Where is the user authentication logic defined?"
  2. **Documentation Embeddings:** The official documentation for key libraries (FastAPI, React, Pydantic) is indexed. This grounds the agent in accurate syntax, reducing hallucinations about non-existent API methods.26
  3. **Learned Solutions:** When the agent successfully solves a difficult bug, the solution is summarized and stored here. Future agents can retrieve this solution via RAG when encountering similar error signatures.

### 8.3 Reinforcement Learning Signal

In the PoC phase, we use a heuristic-based reward signal to guide the agent, approximating a simplified RLHF process.

* **Reward Function:**
  + **+1.0:** Code compiles, passes all tests, and passes security scan.
  + **+0.5:** Code compiles and passes tests, but fails style check (Lint).
  + **-0.5:** Code compiles but fails tests.
  + **-1.0:** Code fails to compile (Syntax Error).
* **Mechanism:** This signal is used to prioritize successful "plans" in the Semantic Memory. Successful trajectories are weighted higher during retrieval.

## 9. Implementation Roadmap and Timeline

The development of the AI SWE is structured into five aggressive phases over a 10-week timeline.1

### Phase 1: Foundation and Sandboxing (Weeks 1-2)

* **Objective:** Establish the secure runtime environment and basic tool usage.
* **Activities:**
  + Set up **Ollama** with **Qwen2.5-Coder 7B**.
  + Develop the **Docker** sandbox with Python/Node.js runtimes.
  + Implement basic "Agent" class with write\_file and run\_shell tools.
* **Deliverable:** A script that can take a prompt, generate a Python file, run it in Docker, and return the output.

### Phase 2: Orchestration and State Management (Weeks 3-4)

* **Objective:** Implement the "Brain" of the system using LangGraph.
* **Activities:**
  + Define the AgentState schema.
  + Implement the **Planner** and **Implementation** agents.
  + Wire the nodes and edges in **LangGraph**.
  + Connect **SQLite** for state persistence.
* **Deliverable:** A CLI tool that can break a requirement into 3 tasks and execute them sequentially.

### Phase 3: The Testing and RLEF Loop (Weeks 5-6)

* **Objective:** Enable self-correction.
* **Activities:**
  + Implement the **Test Agent** and Pytest integration.
  + Build the "Conditional Edge" logic in LangGraph (Pass -> Review, Fail -> Retry).
  + Develop the stack trace parsing logic.
* **Deliverable:** The system can autonomously fix a syntax error or a failing test without human input.

### Phase 4: Memory and RAG Integration (Weeks 7-8)

* **Objective:** Give the agent long-term context.
* **Activities:**
  + Deploy **ChromaDB**.
  + Implement the indexing pipeline for the codebase.
  + Integrate RAG tools into the Planner and Coder agents.
* **Deliverable:** The agent can modify an *existing* file by retrieving it from context, understanding it, and applying a patch.

### Phase 5: Security, UI, and Benchmarking (Weeks 9-10)

* **Objective:** Polish and Validation.
* **Activities:**
  + Integrate **Bandit** and **Semgrep** (Reviewer Agent).
  + Build the Web Dashboard (React) for visualization.
  + Run the system against the **SWE-bench Lite** dataset.
* **Deliverable:** Final PoC demonstration achieving >25% on SWE-bench tasks.1

## 10. Risk Analysis and Mitigation Strategies

| **Risk Category** | **Threat Description** | **Likelihood** | **Impact** | **Mitigation Strategy** |
| --- | --- | --- | --- | --- |
| **Hallucination** | Agent generates code using non-existent libraries or methods. | High | High | **Strict RAG:** Force agent to retrieve docs before using APIs. **RLEF:** Compiler feedback immediately catches syntax errors. |
| **Infinite Loops** | Agent gets stuck in a "Fail-Fix-Fail" cycle, consuming resources. | Medium | Medium | **Circuit Breakers:** Hard limit of 5 retries. **Diversity Sampling:** Force a different approach/plan after 2 consecutive failures.16 |
| **Security (Moltbook)** | Agent exploits the environment or exfiltrates data. | Low | Critical | **Network Isolation:** Sandbox has no internet egress. **Semantic Firewall:** Output scanning. **Least Privilege:** Agent runs as non-root user. |
| **Token Limits** | Long conversations exceed the 32k context window. | High | Medium | **Context Compaction:** Summarize completed tasks in the prompt. **Vector Store:** Offload code storage to ChromaDB, only retrieving relevant chunks. |
| **Model Performance** | Qwen 7B is too small for complex architectural reasoning. | Medium | High | **Hierarchical Planning:** Use the Planner to break tasks into tiny, atomic units that are easy for a 7B model to handle. |

## 11. Success Metrics (KPIs)

To objectively evaluate the success of the PoC, the following metrics will be tracked.1

1. **Autonomous Resolution Rate (ARR):** The percentage of assigned tasks completed *without* any human intervention.
   * **Target:** > 80% for routine tasks (e.g., "Add a field to this API").
2. **Self-Correction Rate (SCR):** The frequency with which the agent successfully repairs a bug it introduced after receiving error feedback.
   * **Target:** > 60%.
3. **SWE-bench Lite Score:** Performance on the industry-standard benchmark for software engineering agents.
   * **Target:** > 25% (comparable to a capable Junior Engineer).
4. **Cost Efficiency:** The average compute cost per resolved feature.
   * **Target:** < $0.50 (electricity cost) vs. $50+ (Human hour).
5. **Mean Time to Resolution (MTTR):** The time from Prompt to Deployed Code.
   * **Target:** < 10 minutes for standard features.

## 12. Conclusion

The **Autonomous Self-Evolving Software Engineer (AI SWE)** represents the future of software development. By integrating the reasoning capabilities of **Qwen2.5-Coder** with the stateful orchestration of **LangGraph** and the rigorous feedback loops of **RLEF**, this system moves beyond the limitations of "Copilots." It offers a vision of an engineering organization where human creativity is amplified by a tireless, self-correcting, and infinitely scalable digital workforce. The architecture defined in this PRD provides a concrete, actionable path to realizing this vision, balancing ambition with the practical constraints of security, cost, and technology maturity.

# Detailed Analysis and Second-Order Insights

## 1. The Shift to "Agentic" Architectures: Implications of State

The transition from simple "Copilot" autocomplete to the "AI SWE" represents a fundamental shift in state management.

* **Insight:** Traditional LLM interactions are stateless REST calls—a "fire and forget" mechanism. The AI SWE, via **LangGraph**, introduces *persistent state* to the reasoning process. The system "remembers" that it is in the middle of a debugging cycle, maintaining the history of its hypotheses and failures.
* **Implication:** This requires a shift in how we debug AI. We can no longer just analyze the "prompt" and "response"; we must analyze the entire "trajectory" or "state history" of the agent. This necessitates new observability tools (visualizing the graph state) rather than just log aggregators. The choice of **SQLite** for Episodic Memory is strategic here—it allows for SQL-based querying of agent *behavioral patterns* (e.g., "Select \* from logs where ErrorType='SyntaxError' AND Attempt > 3").
* **Ripple Effect:** This statefulness enables **Long-Horizon Planning**. An agent can start a task, realize it is missing a dependency, pause the current task, spawn a sub-task to install the dependency, and then *resume* the original task exactly where it left off. This capabilities hierarchy is what distinguishes an "Engineer" from a "Coder."

## 2. The Economics of Open-Weight Models (Qwen vs. GPT-4)

The PRD mandates **Qwen2.5-Coder 7B** via **Ollama**. This is a critical economic and strategic decision, not just a technical one.

* **Trend:** The performance gap between "Frontier Models" (GPT-4, Claude 3 Opus) and "Specific Domain Models" (Qwen-Coder, DeepSeek-Coder) is collapsing for specialized tasks like coding.
* **Insight:** By using a 7B model locally, the marginal cost of "thinking" drops to near zero (electricity cost only). This unlocks **"High-Volume Reasoning"**. We can afford to let the agent "think" for 10 minutes, generating 100 variations of a plan, simulating them, and picking the best one. This would be cost-prohibitive with GPT-4 API calls (potentially costing dollars per task).
* **Second-Order Effect:** This enables "Brute Force Creativity." The agent can generate extensive test suites covering thousands of edge cases, running them all, and only keeping the code that passes. Quantity of compute substitutes for quality of single-shot insight. It fundamentally changes the economics of software testing from "expensive and manual" to "cheap and exhaustive."

## 3. The "Junior Engineer Paradox" and Organizational Impact

The AI SWE is explicitly positioned as a "Junior Engineer" persona.

* **Contradiction:** The AI has the *knowledge* of a senior engineer (access to all docs/patterns in ChromaDB) but the *judgment* of a junior (liable to get stuck in loops, miss subtle business context, or over-engineer solutions).
* **Insight:** The deployment of such agents will not replace engineers but will **bifurcate** the engineering role. "Coding" (syntax generation) becomes a commodity, devalued to near zero. The value shifts entirely to **"Specification"** (writing the PRD) and **"Review"** (verifying the output).
* **Ripple Effect:** Organizations will need to rethink their hiring and training pipelines. We may see a "hollowed-out" middle, where fewer juniors are hired because agents do that work. However, this creates a crisis: how do you create Senior Engineers if no one does the Junior work? The AI SWE effectively *is* the new Junior Developer pipeline. Humans must start as "Agent Supervisors," learning to debug the agent's logic rather than writing the code themselves. The "Input" to the engineering process becomes natural language prose (the PRD), making written communication skills paramount for future engineers.

## 4. Security as a Deterministic Constraint in a Probabilistic System

The inclusion of **Bandit** and **Semgrep** highlights a crucial limitation of LLMs: they are probabilistic engines.

* **Insight:** You cannot prompt an LLM to "never write insecure code" with 100% certainty. It works on probability distributions; there is always a non-zero chance it will hallucinate a vulnerability.
* **Implication:** Security must be **externalized** from the model. It must be a hard, deterministic gate (the Reviewer Agent tools). The "Hybrid Architecture" (Probabilistic GenAI + Deterministic Static Analysis) is the only viable path for enterprise AI. We rely on the AI for creativity and speed, but we rely on deterministic code for safety.
* **Risk:** The "Moltbook" scenario (where agents communicate unchecked) drives the absolute requirement for **Docker** sandboxing. The agent is treated as an "untrusted insider threat" by default. We must assume the agent *will* try to run malicious code (either by accident or prompt injection) and architect the environment to contain it.

## 5. RLEF: The Engine of Improvement

Reinforcement Learning from Execution Feedback (RLEF) is the technical heart of the system.

* **Mechanism:** It converts the "Compiler" and "Test Runner" into a "Reward Function."
* **Insight:** This solves the "Hallucination" problem structurally. If the code doesn't run, the agent gets a negative signal immediately. It is grounded in reality.
* **Future Outlook:** As the Episodic Memory (SQLite) grows, the agent builds a private "Stack Overflow" of solutions that worked *in this specific project context*. Over time, the agent becomes "specialized" to the company's specific codebase and architectural quirks, increasing its value the longer it runs. This creates a **Data Moat** that generic models cannot replicate. A generic GPT-4 model doesn't know that *your* legacy database requires a specific, weird connection string. Your AI SWE *does* know that, because it failed 5 times last month and learned the fix.

## 6. Infrastructure Complexity & Maintenance

* **Observation:** The stack (Docker, Kubernetes, LangGraph, Ollama, Vector DB, SQL DB) is complex.
* **Insight:** **"AgentOps"** becomes a new discipline. Maintaining the *agent's environment* (ensuring Docker runs, vector indices are fresh, Ollama is up) becomes as critical as maintaining the application servers.
* **Requirement:** This justifies the **Runtime Agent** (SRE persona) – the AI eventually needs to help maintain *itself* and its environment, closing the final loop of autonomy. We are building systems that build systems.

# Technical Appendix

## A. Agent Prompt Templates (System Instructions)

### 1. Planner Agent

SYSTEM: You are the Chief Architect and Technical Lead.

CONTEXT: We are building a software system based on the user's PRD.

GOAL: Decompose the high-level requirement into a detailed, step-by-step implementation plan.

INSTRUCTIONS:

1. Analyze the requirements for implicit needs (e.g., "Login" -> "DB Schema", "Auth API", "Frontend Form").
2. Create a Directed Acyclic Graph (DAG) of tasks. Identify dependencies (Task B cannot start until Task A is done).
3. Use the "Retrieve\_Architecture" tool to check our standard patterns in ChromaDB.
4. Output strictly valid JSON matching the PlanSchema.

### 2. Implementation Agent

SYSTEM: You are a Senior Polyglot Developer (Python/TypeScript).

CONTEXT: You are executing Task ID: {task\_id} - "{task\_description}".

INPUT: {codebase\_context}

TOOLS: FileSystem, Docker, Python 3.11.7.

INSTRUCTIONS:

1. Write clean, typed, and documented code. Follow PEP8 (Python) or ESLint (JS) standards.
2. DO NOT use placeholder comments like "# implement logic here". Write the full implementation.
3. If you need external libraries, you MUST add them to requirements.txt or package.json.
4. Think step-by-step: "First I will create the file, then I will add the imports, then the class..."

### 3. Test Agent

SYSTEM: You are a QA Automation Engineer.

CONTEXT: The Implementation Agent has just written code for Task ID: {task\_id}.

GOAL: Break the code. Verify it conforms to requirements.

INSTRUCTIONS:

1. Generate a Pytest suite for the new code.
2. Include Positive tests (Happy Path), Negative tests (Invalid Inputs), and Edge Cases (Nulls, Boundaries).
3. Execute the tests in the Docker sandbox.
4. If tests fail, analyze the output and provide a structured bug report.

## B. StateGraph Logic (LangGraph Pseudocode)

Python

from langgraph.graph import StateGraph, END  
  
# Define Nodes  
def planner(state):...  
def coder(state):...  
def tester(state):...  
def reviewer(state):...  
  
# Define Logic  
workflow = StateGraph(AgentState)  
workflow.add\_node("planner", planner)  
workflow.add\_node("coder", coder)  
workflow.add\_node("tester", tester)  
workflow.add\_node("reviewer", reviewer)  
  
workflow.set\_entry\_point("planner")  
workflow.add\_edge("planner", "coder")  
workflow.add\_edge("coder", "tester")  
  
# Conditional Logic for Self-Correction  
def check\_test\_results(state):  
 if state['test\_results'] == 'PASS':  
 return "reviewer"  
 else:  
 # The Self-Healing Loop  
 if state['iteration\_count'] > MAX\_RETRIES:  
 return "reviewer" # Fail open / Escalate to human  
 else:  
 state['iteration\_count'] += 1  
 return "coder" # Loop back to fix based on error logs  
  
workflow.add\_conditional\_edges("tester", check\_test\_results)  
workflow.add\_edge("reviewer", END)  
  
app = workflow.compile()

## C. Database Schemas

**Episodic Memory (SQLModel):**

Python

class AgentLog(SQLModel, table=True):  
 id: Optional[int] = Field(default=None, primary\_key=True)  
 timestamp: datetime = Field(default\_factory=datetime.utcnow)  
 project\_id: str  
 agent\_role: str  
 action\_type: str # "Reasoning", "ToolUse", "Output"  
 input\_prompt: str  
 output\_response: str  
 execution\_success: bool  
 error\_message: Optional[str]  
 duration\_ms: int

**Semantic Memory (ChromaDB Metadata):**

* **Collection:** codebase\_v1
* **Vector:** [0.12, -0.05,...] (1536 dimensions via text-embedding-ada-002)
* **Metadata:**
  + file\_path: src/auth/router.py
  + language: python
  + type: function
  + name: login\_user
  + dependencies: ['users\_db', 'jwt\_handler']
  + last\_modified: 2023-10-27

This report constitutes a complete, actionable blueprint for building the AI SWE. It integrates the specific technologies requested (**Qwen/Ollama/LangGraph**) with the strategic objectives of the uploaded PoC document, providing a rigorous path from concept to autonomous reality.

#### Works cited

1. Autonomous Software Engineer PoC Document.docx
2. How To Define an AI Agent Persona by Tweaking LLM Prompts - The New Stack, accessed on February 5, 2026, <https://thenewstack.io/how-to-define-an-ai-agent-persona-by-tweaking-llm-prompts/>
3. How to Run Qwen2.5 Models Locally in 3 Minutes? - Analytics Vidhya, accessed on February 5, 2026, <https://www.analyticsvidhya.com/blog/2025/01/run-qwen2-5-models-locally/>
4. Qwen2.5 Coder Instruct 7B Intelligence, Performance & Price Analysis, accessed on February 5, 2026, <https://artificialanalysis.ai/models/qwen2-5-coder-7b-instruct>
5. GPT-4 vs Qwen2.5 Coder 7B Instruct (Comparative Analysis) - Galaxy.ai Blog, accessed on February 5, 2026, <https://blog.galaxy.ai/compare/gpt-4-vs-qwen2-5-coder-7b-instruct>
6. AI Agents for Engineering Leadership - Waydev, accessed on February 5, 2026, <https://waydev.co/features/way-ai-agents/>
7. How can different user personas get the best out of an internal developer portal? - Port.io, accessed on February 5, 2026, <https://www.port.io/blog/how-can-different-user-personas-get-the-best-out-of-an-internal-developer-portal>
8. Vibe coding is not the same as AI-Assisted engineering. | by Addy Osmani - Medium, accessed on February 5, 2026, <https://medium.com/@addyosmani/vibe-coding-is-not-the-same-as-ai-assisted-engineering-3f81088d5b98>
9. User story examples: Agile marketing tactics - Adobe for Business, accessed on February 5, 2026, <https://business.adobe.com/blog/basics/user-story-examples>
10. The 2025 AI Agent Security Landscape: Players, Trends, and Risks, accessed on February 5, 2026, <https://www.obsidiansecurity.com/blog/ai-agent-market-landscape>
11. qwen2.5-coder:7b - Ollama, accessed on February 5, 2026, <https://ollama.com/library/qwen2.5-coder:7b>
12. Agentic AI Use Cases That Prove the Power of Intelligent Automation - Moveworks, accessed on February 5, 2026, <https://www.moveworks.com/us/en/resources/blog/agentic-ai-examples-use-cases>
13. Qwen/Qwen2.5-Coder-7B - Hugging Face, accessed on February 5, 2026, <https://huggingface.co/Qwen/Qwen2.5-Coder-7B>
14. Qwen/Qwen2.5-Coder-7B-Instruct - Hugging Face, accessed on February 5, 2026, <https://huggingface.co/Qwen/Qwen2.5-Coder-7B-Instruct>
15. Qwen 2.5 Coder 7B - Open Laboratory, accessed on February 5, 2026, <https://openlaboratory.ai/models/qwen-2_5-coder-7b>
16. LangGraph: Building Self-Correcting RAG Agent for Code Generation, accessed on February 5, 2026, <https://learnopencv.com/langgraph-self-correcting-agent-code-generation/>
17. LangGraph: Agent Orchestration Framework for Reliable AI Agents - LangChain, accessed on February 5, 2026, <https://www.langchain.com/langgraph>
18. Non-functional requirements. Previous… | by Kostiantyn Ivanov | Medium, accessed on February 5, 2026, <https://medium.com/@svosh2/non-functional-requirements-ce6e55305900>
19. Qwen2.5 Speed Benchmark - Qwen - Read the Docs, accessed on February 5, 2026, <https://qwen.readthedocs.io/en/v2.5/benchmark/speed_benchmark.html>
20. Local LLM Speed: Qwen2 & Llama 3.1 Real Benchmark Results - Ajit Singh, accessed on February 5, 2026, <https://singhajit.com/llm-inference-speed-comparison/>
21. Agentic AI Software Engineers: Programming with Trust - arXiv, accessed on February 5, 2026, <https://arxiv.org/html/2502.13767v3>
22. AI Agents, UI Design Trends for Agents | Fuselab Creative, accessed on February 5, 2026, <https://fuselabcreative.com/ui-design-for-ai-agents/>
23. 7 UX Patterns for Designing Trustworthy AI Agents | by VSDesign - Medium, accessed on February 5, 2026, <https://medium.com/@vsdesign37/7-ux-patterns-for-designing-trustworthy-ai-agents-1d903cd7806f>
24. Adaptive: Building Self-Healing AI Agents — A Multi-Agent System for Continuous Optimization | by Madhur Prashant | Medium, accessed on February 5, 2026, <https://medium.com/@madhur.prashant7/evolve-building-self-healing-ai-agents-a-multi-agent-system-for-continuous-optimization-0d711ead090c>
25. Introducing Devin, the first AI software engineer - Cognition, accessed on February 5, 2026, <https://cognition.ai/blog/introducing-devin>
26. AI in Software Development - IBM, accessed on February 5, 2026, <https://www.ibm.com/think/topics/ai-in-software-development>
27. AI Agents: Nonfunctional Requirements That Ensure Reliability - Centizen Inc, accessed on February 5, 2026, <https://www.centizen.com/nonfunctional-requirements-reliable-ai-agents/>