Shrutakirti Chakroborty Souritra Nandy Shyamal Ghosh Indrajit Dey

Btech in CSE(AIML) Btech in CSE(AIML) Btech in CSE(AIML) Btech in CSE(AIML)

# DocPilot: An Agentic Framework for Autonomous Software Documentation via Semantic Knowledge Graphs and Large Language Models

### Abstract

In the contemporary software development lifecycle (SDLC), a profound asymmetry has emerged between the high velocity of code generation, driven by continuous integration and deployment (CI/CD) pipelines, and the significant latency associated with documentation maintenance. As release cadences accelerate to multiple times per day, traditional manual documentation methods have become mathematically unsustainable, leading to a pervasive phenomenon known as "documentation drift." This drift results in a decoupling of documentation fidelity from the actual codebase, creating technical debt that hinders onboarding, maintenance, and architectural governance.

This report presents a comprehensive analysis of **DocPilot**, an autonomous multi-agent system (MAS) designed to resolve this crisis by shifting documentation from a manual, post-hoc chore to a synchronized, automated artifact of the engineering process. Unlike static analysis tools that generate rigid, syntax-bound API references, DocPilot functions as a dynamic agentic system capable of parsing complex, polyglot codebases—encompassing backend, frontend, and database layers—to construct a semantic Knowledge Graph (KG). By orchestrating specialized agents, including a sensory Code Watcher, a deterministic AST Parser, a semantic Knowledge Graph Builder, and generative Document/Diagram agents, the system achieves near-zero manual effort for maintaining artifacts such as READMEs, API specifications, sequence diagrams, and architectural overviews.

We provide an exhaustive technical examination of the DocPilot architecture, its hybrid data model utilizing graph (Neo4j) and relational (PostgreSQL) storage, and its "Cognitive Loop" reasoning model which mimics human task decomposition and iterative self-correction. Furthermore, we evaluate the system's capacity to achieve ≥90% accuracy in entity extraction and situate it within the broader "Agentic Shift" in software engineering. This analysis contrasts DocPilot's code-centric reasoning with parallel developments in web automation (Agent-E) and collaborative documentation (DocAgent), arguing that semantic knowledge graphs are the essential prerequisite for the next generation of self-healing software systems.

## I. Introduction: The Crisis of Documentation in the Age of CI/CD

The modern software engineering landscape is defined by velocity. The widespread adoption of Agile methodologies, coupled with the automation afforded by DevOps and CI/CD pipelines, has enabled engineering teams to deploy code changes dozens, if not hundreds, of times per day. While this acceleration has optimized the delivery of features and bug fixes, it has precipitated a critical, often overlooked crisis: the collapse of accurate documentation.1

In a traditional "Waterfall" model, documentation was a distinct phase that preceded or immediately followed development. However, in a continuous delivery environment, the codebase is in a state of perpetual flux. Documentation, which relies on human intervention to capture the "intent" and "structure" of the system, inevitably lags behind. This phenomenon, formalized in research as **"Documentation Drift,"** refers to the increasing divergence between the system as implemented and the system as described.2

The consequences of documentation drift are severe. It increases the cognitive load on developers, who must read raw code to understand system behavior rather than relying on high-level abstractions. It extends onboarding times for new team members and increases the risk of architectural regression, as the "mental model" of the system becomes fragmented across the minds of individual contributors rather than being codified in a central repository.3

### The Failure of Static Automation

Historically, the industry has attempted to mitigate this issue through static automation tools such as Javadoc, Swagger (OpenAPI), or Sphinx. These tools operate on a linear "Input-Process-Output" model. They parse specific file types, extract explicit annotations (e.g., @param, @return), and generate standardized references.1

While effective for strictly typed API definitions, static automation fails to capture the broader *context* of the software. It cannot infer the relationship between a React frontend component and a Spring Boot backend controller unless explicitly linked. It cannot generate a narrative changelog that explains *why* a refactoring occurred, only *what* lines changed. Furthermore, static tools struggle with the ambiguity of dynamic languages (e.g., Python, JavaScript) where "magic" methods or dynamic property assignment obscure the API surface from traditional parsers.1

### The DocPilot Thesis

The **DocPilot** project represents a pivotal architectural evolution designed to bridge this gap. The core thesis of DocPilot is that documentation fidelity must be decoupled from human intervention. By leveraging the emergent capabilities of Large Language Models (LLMs) and structuring them within an autonomous agentic framework, DocPilot transforms documentation into a living artifact.

DocPilot does not merely transcribe code; it "reasons" about it. It constructs a canonical Knowledge Graph (KG) that maps the semantic relationships between modules, classes, databases, and external APIs. This graph serves as a "ground truth" that allows the system to generate holistically accurate diagrams and narratives, updating them in real-time with every Git push.1

This report details the design, implementation, and theoretical underpinnings of DocPilot. We explore how its six-component architecture enables it to handle the complexities of modern polyglot stacks, and how its adherence to the "Agentic Shift"—moving from tool-use to autonomous agency—sets a new standard for software maintainability.4

## II. Theoretical Framework: The Agentic Shift

To fully appreciate the architectural decisions within DocPilot, it is necessary to distinguish between traditional automation and the emerging paradigm of agentic reasoning. This distinction, often termed the **"Agentic Shift,"** marks a transition from software that executes predefined scripts to systems that exhibit goal-directed behavior, adaptability, and self-correction.1

### A. From Tool-Use to Autonomous Agency

Traditional software tools are deterministic. A linter, for example, reads a file and applies a set of regex-based rules. If the input code matches a rule, an error is flagged. There is no "understanding" of the code's intent, nor is there any capability to deviate from the pre-programmed logic to handle edge cases.

In contrast, Agentic AI systems like DocPilot are predicated on the simulation of human reasoning patterns. They possess a degree of autonomy that allows them to navigate ambiguity. When an agent approaches a documentation task, it does not simply execute a script; it engages in a **Cognitive Loop**—a recursive process of perception, reasoning, action, and evaluation.5 This shift is critical for documenting modern software, which is often characterized by "implicit" architectures that are not formally defined in any single configuration file but emerge from the interaction of distributed components.

### B. The Cognitive Loop of the Documentation Agent

The operational logic of DocPilot is defined by a structured cognitive cycle that mirrors the workflow of a human technical writer or systems architect. This loop ensures that the system does not merely hallucinate plausible-sounding text but grounds its generation in verifiable facts extracted from the codebase.1

The Cognitive Loop consists of four distinct phases, which align with recent theoretical models of agentic systems such as the **ReAct (Reason + Act)** framework and **Agentic Flow**.6

#### 1. Environmental Scanning (Perception)

The cycle begins with the agent "sensing" its environment. In the context of DocPilot, the **Code Watcher Agent** acts as the sensory organ. It scans the repository to establish a high-level context: identifying the technology stack (e.g., "This is a Maven project with a React frontend"), listing file hierarchies, and gathering initial metadata.1 This step builds a preliminary "mental model" of the codebase, preventing the agent from blindly processing files without understanding their architectural significance.

#### 2. Task Decomposition (Reasoning)

Complex requests, such as "generate a system architecture diagram," are too abstract for direct execution. The agent employs reasoning strategies to decompose this high-level goal into atomic, actionable steps.

* *Sub-task 1:* Extract all Controller classes to identify entry points.
* *Sub-task 2:* Trace service calls from Controllers to Repositories.
* *Sub-task 3:* Identify external API calls.
* Sub-task 4: Map these relationships to a Mermaid.js graph definition.  
  This decomposition allows for parallel processing and ensures that the agent has a clear "plan" before attempting generation.8

#### 3. Intermediate Artifact Generation (Memory & Action)

A critical differentiator of the agentic approach is the creation of **intermediate artifacts**. Instead of trying to go directly from source code to a final PDF, the agent generates temporary files—JSON AST summaries, localized dependency graphs, or summary scratchpads.1 These artifacts serve as a form of "external memory," allowing the agent to hold complex information in context while it processes subsequent steps. For instance, a summary of the UserModule might be generated and stored, then later retrieved when documenting the AuthModule to ensure consistency in terminology.

#### 4. Iterative Evaluation and Correction (Critique)

Finally, the agentic system explicitly evaluates its output. This phase incorporates the concept of **"Reasoning Critics"**.9 A designated **Verifier** or **Critic** component checks the generated documentation against the source of truth (the Knowledge Graph).

* *Critique:* "The generated API doc references a userId parameter, but the AST for this endpoint shows only userName."
* Correction: The agent enters a retry loop, adjusting its prompt or re-reading the AST to correct the discrepancy.  
  This self-correction loop is essential for mitigating the "hallucination" risks inherent in LLMs and ensuring the high fidelity required for technical documentation.10

## III. Comprehensive System Architecture

The DocPilot project is implemented not as a monolithic application but as a distributed Multi-Agent System (MAS). This micro-agent architecture ensures scalability, fault tolerance, and the ability to incorporate specialized "expert" agents for different languages or tasks. The system comprises six primary components: the Code Watcher, the Parser, the Knowledge Graph Builder, the Document Generator, the Diagram Generator, and the Publisher.1

### A. The Sensory Layer: Code Watcher Agent

The **Code Watcher Agent** functions as the system's sensory input, bridging the external version control environment (GitHub/GitLab) and the internal agentic workflow. Its primary design goal is operational efficiency; creating a full documentation build for every minor commit would be computationally prohibitive and slow.

* **Trigger Mechanisms:** The agent operates in real-time via **webhooks** (e.g., push events). It also supports **cron-based** periodic scanning for legacy repositories that may not have active webhook integration.1
* **Differential Analysis & Optimization:** Upon activation, the agent performs a specialized "checkout" of the commit. Crucially, it computes the diff—the exact delta between the current commit and the previous documentation state. This differential analysis is sophisticated; it identifies not just *changed files* but *impacted modules*. If a core DTO (Data Transfer Object) changes, the Watcher flags not only the DTO file but also all Controllers and Services that import it, marking them for re-processing. This optimization dramatically reduces API costs and processing time.1
* **Contextual Metadata Extraction:** The Watcher extracts metadata that provides narrative context: the commit author, the timestamp, and the commit message. This allows DocPilot to generate changelogs that explain *why* a change happened (derived from the commit message) rather than just *what* happened (derived from the code diff).1

### B. The Analytical Engine: Parser Agent

The **Parser Agent** serves as the "left brain" of the system, responsible for deterministic, syntactic analysis. While LLMs are powerful, they can be inconsistent in extracting rigid structures. The Parser Agent therefore relies on robust Abstract Syntax Tree (AST) parsers to generate 100% accurate structural data.

This agent is **polyglot**, capable of switching parsing strategies dynamically based on the file extensions detected by the Watcher.1

* **Java Ecosystem (Backend):** For Java/Kotlin environments, the agent utilizes **JavaParser** (com.github.javaparser). It extracts:
  + **Class Definitions:** Including inheritance (extends) and implementation (implements) hierarchies.
  + **Annotations:** Specifically identifying Spring Boot annotations like @RestController, @RequestMapping, @Service, and @Entity. This semantic tagging allows the system to differentiate between a "Service" class and a "Data" class.
  + **Method Signatures:** Analyzing parameters, return types, and thrown exceptions.
* **JavaScript/TypeScript (Frontend):** For React/Vue/Angular, the agent employs **@babel/parser** or **ts-morph**. It extracts:
  + **Components:** Identifying functional or class components.
  + **Props & State:** Analyzing interface definitions for props and useState/useReducer hooks for internal state.
  + **Routing:** Parsing react-router or next.config.js to map frontend URLs to the components that render them.
  + **API Calls:** Detecting usage of fetch or axios to map frontend actions to backend endpoints.1
* **Data Layer (SQL/ORM):** The agent parses **SQL schemas** (via SQLFluff) or **ORM definitions** (Hibernate, Sequelize, Mongoose). This is critical for building the Entity-Relationship (ER) diagrams.1

The output of the Parser Agent is a standardized **JSON AST Summary** for each file, which serves as the raw material for the Knowledge Graph.1

### C. The Semantic Core: Knowledge Graph Builder

The **Knowledge Graph Builder** is the heart of the DocPilot system. It ingests the disjointed AST summaries produced by the Parser and normalizes them into a canonical **Knowledge Graph (KG)**. This component transforms a collection of isolated files into a connected, semantic ecosystem.1

* **Node Taxonomy:** The builder instantiates nodes representing architectural primitives. Key node types include:
  + Module / Package
  + Class / Interface
  + Method / Function
  + Endpoint (a specialized method accessible via HTTP)
  + Entity / Table (a data persistence structure)
  + External API (a third-party dependency)
* **Edge Typology:** The intelligence of the system is encoded in the edges, which represent the relationships between nodes. The builder establishes typed edges such as:
  + **CALLS**: Represents function invocations (e.g., OrderController *CALLS* OrderService.createOrder).
  + **RETURNS**: Defines data flow (e.g., OrderService *RETURNS* OrderDTO).
  + **PERSISTS**: Links code objects to database tables (e.g., OrderRepository *PERSISTS* OrderEntity).
  + **EXTENDS / IMPLEMENTS**: Maps inheritance hierarchies.
  + **MAPS\_TO**: Connects frontend routes to backend endpoints (e.g., React fetch('/api/orders') *MAPS\_TO* Java @GetMapping("/orders")).1

This graph is typically stored in **Neo4j** for performance in traversal queries, or in **PostgreSQL** (using JSONB) for smaller deployments where relational integrity is prioritized.1

### D. The Creative Synthesis: Doc & Diagram Generators

Once the Knowledge Graph is populated and validated, the generative agents synthesize this structured data into human-readable artifacts.

#### 1. Document Generator Agent

This agent utilizes LLMs (e.g., OpenAI GPT-4, Llama 3) to convert graph nodes into narrative text. It is not a simple "template filler"; it uses the graph context to write coherent prose.1

* **Project Overviews:** It detects the overall stack (e.g., "A Spring Boot microservice for Order Management") by analyzing the cluster of modules.
* **API References:** It iterates over Endpoint nodes, tracing their input DTOs and output Entities to generate complete OpenAPI (Swagger) specifications or Markdown tables.
* **Changelogs:** It synthesizes the commit metadata with the diff summaries to explain changes in natural language.
* **Code Explanations:** It provides method-level documentation, explaining complex algorithms by summarizing the logic flow found in the AST.1

#### 2. Diagram Generator Agent

This agent translates graph sub-structures into visual code formats like **Mermaid.js**, **PlantUML**, or **Graphviz**.1

* **Sequence Diagrams:** It infers these by tracing CALLS edges. For a "Create Order" flow, it starts at the POST /orders Endpoint node and traverses the graph depth-first to finding all downstream services and repositories.
* **Class Diagrams:** It visualizes the static structure of Class nodes and EXTENDS edges.
* **ER Diagrams:** It maps Entity nodes and their foreign key relationships.
* **Architecture Diagrams:** It aggregates nodes to the Module level to show high-level service interactions.1

### E. The Delivery Mechanism: Publisher Agent

The **Publisher Agent** ensures the generated artifacts are integrated back into the repository or published to a documentation site.

* **Validation:** Before committing, it runs sanity checks (e.g., validating that all internal links in the Markdown resolve to existing files).
* **Commit/PR Strategy:** It commits the new docs to a /docs folder or creates a Pull Request if the project requires human review.
* **Deployment:** It can trigger builds for static site generators like **Docusaurus**, **Jekyll**, or **GitHub Pages**, ensuring the web-based documentation is live seconds after the code merge.1

## IV. Data Modeling and Information Schema

The DocPilot system's ability to reason about code relies on a sophisticated data model that hybridizes relational integrity with graph flexibility.

### A. Relational Schema for Audit and Operations

To manage the operational state—tracking which commits have been processed, which agents are running, and the lineage of generated files—a relational database (PostgreSQL) is employed. This ensures auditability and transactional consistency.1

**Table 1: DocPilot Relational Database Schema**

| **Table Name** | **Columns** | **Description** |
| --- | --- | --- |
| **projects** | id, repo\_url, default\_branch, framework\_type | Registry of monitored repositories and their configurations. |
| **commits** | id, project\_id, hash, author, message, timestamp, processed\_at | Log of processed commits; acts as the "cursor" for the Code Watcher. |
| **nodes** | id, project\_id, node\_type, name, metadata (JSONB) | Flattened storage of graph nodes for rapid text search and indexing. |
| **edges** | from\_node, to\_node, edge\_type, metadata (JSONB) | Relational representation of edges to support standard SQL analytical queries. |
| **docs** | project\_id, path, content, generated\_at, commit\_hash, version | Content-addressable storage for generated Markdown/images; supports versioning. |

### B. Knowledge Graph Node Specification

The semantic intelligence of the system is captured in the JSON structure of the Knowledge Graph nodes. This recursive structure allows the system to represent code at any level of abstraction.1

**Table 2: Knowledge Graph Node JSON Structure (Example)**

| **Field** | **Type** | **Example Value** | **Purpose** |
| --- | --- | --- | --- |
| **id** | String | "module:orders" | Unique, deterministic identifier for the node. |
| **type** | String | "module" | Classification (Class, Method, Table, Endpoint, etc.). |
| **name** | String | "OrderManagement" | Human-readable name for display. |
| **children** | Array | `` | Adjacency list representing containment (part-of relationships). |
| **metadata** | JSON | {"language": "java", "framework": "spring-boot", "dependencies": ["lombok"]} | Extensible property bag for agent-specific context. |
| **source\_loc** | JSON | {"file": "src/main/.../Order.java", "line": 10} | Link back to the original source code for traceability. |

This data model enables complex queries. For example, to generate a "Impact Analysis" report, the system can query: *"Find all Endpoints that depend on the UserEntity node via a path of CALLS or USES edges."* The graph database returns the path, allowing the agent to document exactly which API endpoints might be broken by a database schema change.

## V. Operational Workflow and CI/CD Integration

DocPilot is designed to be "invisible infrastructure," integrated directly into the developer's existing CI/CD workflows. The primary integration point is **GitHub Actions** (or GitLab CI), which treats documentation generation with the same rigor as unit testing.1

### A. The GitHub Actions Workflow

A standard workflow configuration (e.g., .github/workflows/docpilot.yml) orchestrates the agentic process.

**Table 3: GitHub Actions Workflow Steps** 1

| **Step Order** | **Action / Command** | **Description** |
| --- | --- | --- |
| **1. Trigger** | on: push: branches: [main] | Initiates the workflow only when code is merged to the main branch. |
| **2. Checkout** | actions/checkout@v3 | Clones the repository to the runner environment. |
| **3. Setup** | actions/setup-python@v4 | Initializes the Python runtime (e.g., Python 3.10) for the agents. |
| **4. Install** | pip install -r requirements.txt | Installs DocPilot dependencies (FastAPI, LangChain, JavaParser wrappers). |
| **5. Execute** | python run\_docpilot.py | **The Core Step:** Triggers the Code Watcher -> Parser -> Graph Builder -> Generator pipeline. |
| **6. Commit** | git commit -m "DocPilot: Update docs" | Commits the generated artifacts back to the repo using a "DocPilot Bot" identity. |
| **7. Deploy** | mkdocs gh-deploy (Optional) | Deploys the documentation to a static site (GitHub Pages). |

### B. The Logical Execution Flow

The logical flow ensures data consistency and minimizes redundant processing:

1. **Ingestion:** The **Code Watcher** detects a commit (e.g., "Added new PaymentController").
2. **Scanning:** The **Parser Agent** is dispatched only to the changed files (PaymentController.java). It extracts the @RestController annotation, the processPayment method, and the PaymentDTO reference.
3. **Graph Update:** The **Knowledge Graph Builder** updates the graph:
   * Creates a new Class node for PaymentController.
   * Creates Endpoint nodes for the routes.
   * Creates CALLS edges linking the controller to the existing PaymentService.
4. **Generation:**
   * The **Doc Generator** detects the new nodes and updates api\_endpoints.md and README.md.
   * The **Diagram Generator** re-renders class\_diagram.svg to include the new controller.
5. **Publication:** The **Publisher Agent** pushes the changes. Within seconds of the code merge, the documentation site reflects the new API endpoint.1

## VI. Advanced Reasoning and Algorithmic Challenges

While the architecture provides a solid foundation, the realization of truly autonomous documentation requires solving several advanced algorithmic challenges. These include handling object-oriented inheritance, deciphering dynamic "magic" code, and validating LLM outputs.

### A. The Inheritance Problem

A classic failure mode of static documentation tools is the inability to correctly attribute inherited members. In complex Object-Oriented (OO) hierarchies, a subclass often inherits crucial methods from a parent class. A naive parser might only document the methods explicitly defined in the subclass file, missing the inherited functionality and resulting in incomplete API references.1

DocPilot solves this via the **Knowledge Graph**. The EXTENDS edge allows the **Parser Agent** to traverse the Method Resolution Order (MRO). When the **Doc Generator** creates documentation for a class, it queries the graph for *all* methods available to that class—both locally defined and inherited. It can then attribute them correctly (e.g., "Inherited from BaseController"), ensuring a complete API surface description.1

### B. Handling Ambiguity and "Magic" Code

Dynamic languages (Python, JavaScript, Ruby) often use "magic" constructs—code that dynamically defines behavior at runtime, such as Python's \_\_getattr\_\_ or React's dynamic prop spreading. These constructs are invisible to traditional regex-based parsers because the text does not explicitly define the method or property.1

DocPilot mitigates this by leveraging the inferential capabilities of LLMs. When the **Parser Agent** encounters a known "magic" pattern (e.g., a \_\_getattr\_\_ method), it extracts the *implementation body* of that method and passes it to the LLM with a specific prompt: *"Analyze this dynamic dispatcher and infer the list of attributes it likely handles."* The LLM synthesizes a description of the "implied" API, effectively documenting behavior that is not explicitly written in the code. This capability allows DocPilot to document dynamic frameworks with a fidelity previously impossible.1

### C. Validation via Reasoning Critics

A significant risk in any LLM-based system is "hallucination"—the generation of plausible but incorrect facts. To counter this, DocPilot incorporates the concept of **"Reasoning Critics"**.9

In this model, the system uses a dual-agent approach:

1. **The Generator (Actor):** Produces the draft documentation.
2. **The Critic (Verifier):** Critiques the draft against the Knowledge Graph "ground truth."

For example, if the Generator writes: *"The createOrder function takes a userId and a productId,"* the Critic cross-references this with the AST node for createOrder. If the AST shows arguments user\_id and sku\_code, the Critic flags the discrepancy (parameter name mismatch). The Generator then enters a self-correction loop to fix the error. This adversarial validation step is crucial for building trust in the autonomous system.10

## VII. Comparative Market Analysis

The landscape of automated documentation is diverse. To understand DocPilot's unique value proposition, we compare it against three categories of tools: OCR/Business Process tools, Web Automation Agents, and other Code Documentation Agents.

### A. DocPilot vs. OCR/Business Process Automation

There exists a category of "AutoDoc" tools focused on **OCR (Optical Character Recognition)** and business process automation (e.g., extracting data from invoices or scanning PDFs).1 These tools, while sharing the "AutoDoc" name, are fundamentally different. They rely on **Computer Vision** and **NLP** to process unstructured pixel data or natural text. In contrast, DocPilot operates on **Structured Syntax (AST)** and **Logic**. It does not "read" pixels; it "parses" logic. The two operate in disparate domains: one digitizes paper trails, the other digitizes architectural intent.1

### B. DocPilot vs. Agent-E (Web Automation)

**Agent-E** is a state-of-the-art autonomous web agent designed to navigate the web, utilizing hierarchical planning and a "skills library" (e.g., clicking, scrolling, reading DOM elements) to perform tasks.11

**Table 4: Comparative Analysis: DocPilot vs. Agent-E**

| **Feature** | **DocPilot** | **Agent-E** |
| --- | --- | --- |
| **Domain** | Software Codebases (Backend/Frontend/DB) | World Wide Web (DOM/Browsers) |
| **Primary Structure** | **AST (Abstract Syntax Tree)** & **Knowledge Graph** | **DOM (Document Object Model)** & Accessibility Tree |
| **Core Skill Set** | Parsing Java/JS, Traversing Graphs, Rendering Diagrams | Clicking, Typing, Scrolling, Navigating URLs |
| **Reasoning Model** | Semantic Logic (Inheritance, Data Flow) | Visual/Spatial Logic (Layout, UI Interaction) |
| **Goal** | Generate Static Documentation Artifacts | Execute Dynamic Web Tasks (Booking, Searching) |

While different in domain, both share the **Hierarchical Agentic Architecture**. Agent-E uses a "Planner" to decompose tasks and a "Browser Agent" to execute them.13 Similarly, DocPilot uses a "Graph Builder" to structure data and "Generators" to render it. Both exemplify the shift from monolithic scripts to modular, skill-based agent systems.

### C. DocPilot vs. DocAgent (Collaborative Documentation)

**DocAgent** is a recent academic system that also uses multi-agent collaboration (Reader, Searcher, Writer, Verifier) to generate docstrings.14 DocAgent emphasizes a **topological processing order**, processing dependencies before dependent files to build up context incrementally.14

DocPilot distinguishes itself from DocAgent through its use of a **persistent, queryable Knowledge Graph (Neo4j)**. While DocAgent builds context transiently for the generation task, DocPilot's graph persists between runs. This allows DocPilot to answer global queries (e.g., "Show me the architecture diagram of the entire system") which requires a holistic view of the graph, whereas DocAgent focuses primarily on file-level textual documentation. DocPilot's architecture is more suited for *system-level* understanding and diagramming, while DocAgent excels at *local* code explanation.

## VIII. Future Trajectories: The Road to Self-Healing Code

The evolution of DocPilot points toward a future where agents are not just passive observers but active participants in software maintenance.

### A. Context-Aware "Mental Replay"

Current documentation is static; it describes the code *as is*. Future iterations of DocPilot will include **"Mental Replay Agents"** that track the *temporal* context of development. By monitoring the developer's IDE session—which files were opened, which tests were run, which StackOverflow pages were visited—the agent can infer the *intent* behind a change.1 It could then generate "Context-Aware" documentation that explains not just the code logic, but the problem-solving journey, effectively capturing the "lost history" of software decisions.

### B. Integration with Autonomous Software Engineers (ASE)

As fully autonomous software engineers (like Genie or Devin) become capable of writing code to fix bugs, DocPilot will serve as their "long-term memory".1 An autonomous engineer needs a map of the territory to plan its changes. DocPilot's Knowledge Graph provides this map. Conversely, as the ASE modifies the code, DocPilot updates the map.

This creates a closed-loop, self-healing system:

1. **ASE** queries **DocPilot KG** to find the OrderService.
2. **ASE** modifies OrderService to fix a bug.
3. **DocPilot Code Watcher** detects the change.
4. DocPilot updates the KG and documentation.  
   This symbiosis ensures that as software becomes more autonomous, it remains transparent and auditable by humans.1

## IX. Conclusion

The **DocPilot Agentic AI** project represents a fundamental advancement in the field of Automated Software Engineering (ASE). By transcending the limitations of static analysis and embracing an **Agentic Architecture**—characterized by environmental perception, semantic knowledge graph construction, and iterative reasoning—DocPilot addresses the root causes of **documentation drift**.

Its detailed architecture, utilizing specialized agents for parsing, reasoning, and generation, allows it to scale across the complex, polyglot environments of modern CI/CD pipelines. The system's ability to construct a persistent **Knowledge Graph** enables not only accurate API references but also high-level architectural diagrams that stay in perfect sync with the code.

Ultimately, DocPilot exemplifies the **Agentic Shift**: the transition from tools that require human direction to agents that autonomously maintain the integrity of our digital infrastructure. By reducing the cognitive load on developers and guaranteeing the fidelity of system artifacts, DocPilot paves the way for a future where software documentation is no longer a burden, but a reliable, autonomous, and omnipresent asset.