DRAKON Diagram Intelligence Enhancement: A Strategic Technical Report

Executive Summary

This report presents a comprehensive research analysis and strategic blueprint for the enhancement of the existing automated code-to-DRAKON-diagram system. The current system, while functional, produces overly simplistic diagrams that fail to capture the semantic depth and control flow complexity of modern codebases. The primary objective is to evolve this tool into a sophisticated, AI-enhanced system capable of generating detailed, accurate, and readable DRAKON diagrams that serve as a true representation of the underlying code logic.

The research yields several key findings that form the basis of the recommended architecture. First and foremost, the critical issue of diagram validity and compatibility with the official DRAKON ecosystem can be definitively solved. Direct server-side use of the official drakonwidget.js library is infeasible due to its inherent dependency on a browser's Document Object Model (DOM). However, a robust and highly recommended solution involves creating a dedicated Node.js microservice that programmatically controls the widget within a headless browser environment (e.g., Puppeteer). This approach guarantees 100% format compatibility and future-proofs the system against changes in the DRAKON specification. Furthermore, analysis of the desktop DRAKON Editor reveals that its native .drn file format is a standard SQLite database, providing a high-performance alternative for direct diagram generation from the Python backend.

To address the core problem of diagram simplicity, this report recommends a multi-layered code analysis engine that moves beyond basic Abstract Syntax Tree (AST) parsing. The foundation of this new engine will be the Control Flow Graph (CFG), which explicitly models all possible execution paths, including complex conditional logic, nested loops, and, crucially, exception handling (try-catch-finally blocks). Layered atop the CFG, Data Flow Analysis (DFA) will be employed to track the lifecycle and state transitions of key variables, providing the

semantic depth currently missing.

Artificial intelligence is integral to achieving the desired level of sophistication. A hybrid AI architecture is proposed, leveraging the complementary strengths of Graph Neural Networks (GNNs) and Transformer-based Large Language Models (LLMs). GNNs will operate on the CFG and DFA representations to identify high-level structural patterns, such as design patterns (e.g., State, Strategy) and anti-patterns (e.g., callback hell). Concurrently, a model like CodeT5, specifically designed for code understanding and generation, will be used to generate human-readable, context-aware labels for diagram nodes by summarizing code blocks and extracting developer intent from variable names and comments.

The final recommendation is a phased implementation plan designed to deliver value incrementally while managing technical risk.

- **Phase 1** focuses on building the headless browser generation service, immediately solving the compatibility and correctness problem.
- **Phase 2** implements the deep program analysis engine based on CFGs and DFAs to achieve the desired level of diagram detail.
- **Phase 3** layers on the AI capabilities for intelligent pattern recognition and semantic node summarization.

This strategic enhancement is projected to deliver a transformative improvement over the current system. It will increase the average node count for complex functions by a factor of 4x to 6x, achieve over 95% coverage of all control flow paths, and, through the headless widget architecture, guarantee perpetual compatibility with official DRAKON editors. The resulting system will not just be a code visualizer but a powerful tool for program comprehension, debugging, and documentation.

The Critical Path: Headless drakonwidget Integration and Format Strategy

The most critical architectural decision for the long-term success and maintainability of the diagram generation system is ensuring that its output is perpetually valid and compatible with the official DRAKON ecosystem, including the drakonwidget.js library, DrakonHub, and the desktop DRAKON Editor.[1] The current approach of manually constructing JSON objects is brittle and prone to breaking as the official format evolves. This section analyzes the feasibility of programmatic generation and proposes a robust, multi-faceted strategy.

Feasibility Analysis of Direct API Usage

An initial investigation focused on the possibility of using the drakonwidget.js library directly within the existing Python backend, potentially through a JavaScript execution environment like PyExecJS. However, a detailed analysis of the library's documentation and design reveals that this approach is fundamentally infeasible.

The drakonwidget API is architected as a client-side, browser-based rendering component, not a server-side generation toolkit.[1] Key functions such as render(width, height, config) are explicitly designed to create and return a DOM element, which the host application must then append to the web page's document structure (e.g., editorArea.appendChild(widgetElement)).[1] The library's configuration options make numerous references to browser-specific technologies, including CSS font definitions (branchFont, canvasLabels) and the HTML5 Canvas 2D context for icon rendering.[1]

These dependencies on a browser environment (specifically, the window object, a complete DOM, and the Canvas API) mean that the library cannot execute in a standard server-side Node.js or PyExecJS environment, which lacks these components. The window.DrakonTestAPI mentioned in the initial query is a client-side testing hook exposed on the window object and is similarly unavailable in a server context. Therefore, any attempt at direct integration would fail due to missing environmental dependencies.

The Headless Browser Architecture: A Robust Solution

While direct integration is not possible, a highly effective and recommended solution is to run drakonwidget.js in the environment it was designed for—a web browser—and to automate this process on the server side using a headless browser. Libraries such as Puppeteer (which controls headless Chrome/Chromium) and Playwright (which supports multiple browsers) are industry-standard tools for this purpose.[3]

The proposed architecture involves creating a dedicated Node.js microservice that acts as a "diagram generation factory." This service decouples the complex task of code analysis from the specific mechanics of diagram creation, leading to a more modular and maintainable system.

The workflow for this architecture would be as follows:

1. **Service Initialization:** The Node.js service starts and maintains a pool of headless browser instances (e.g., using Puppeteer). Each instance loads a minimal local HTML

page that includes the drakonwidget.js script.
2. **API Request:** The Python analysis backend, after processing the source code, sends a structured, high-level description of the desired diagram (e.g., a list of nodes, their types, content, and connections) to a REST API endpoint on the Node.js service, such as /generate.
3. **Programmatic Construction:** The Node.js service receives the request and, within the context of a headless browser page, executes a JavaScript payload. This payload makes a series of calls to the drakonwidget internal API (e.g., createDiagram, addNode, createNewItem) to programmatically construct the diagram in the browser's memory. Because this occurs within a full browser environment, all DOM and Canvas dependencies are met.
4. **JSON Extraction:** Once the diagram is constructed, the service executes another command to call an internal drakonwidget function that serializes the in-memory diagram model into a JSON string.
5. **Validation and Response:** The resulting JSON is inherently valid and correctly formatted because it was generated by the official library's own logic. The service then returns this guaranteed-compatible JSON to the Python backend.

This microservice approach respects the native environments of each component. The Python backend can focus on its core competency of sophisticated code analysis, while the Node.js service handles the specialized task of interacting with a browser-centric JavaScript library. This separation of concerns avoids the creation of a brittle and complex system trying to mock a browser environment in Python.

Fallback and High-Performance Strategy: Direct.drn File Generation

Further research into the DRAKON ecosystem revealed a powerful alternative that offers significant performance advantages. The native .drn file format used by the official desktop DRAKON Editor is not a proprietary binary format but a standard **SQLite 3.6 database**.[5] The complete database schema, including tables such as info, diagrams, items, and tree_nodes, is publicly documented.

This discovery enables a second, parallel implementation path: generating .drn files directly from the Python backend using standard SQLite libraries. This method completely bypasses the need for JavaScript execution or a headless browser, eliminating the associated performance overhead. For large-scale batch processing of thousands of files, this could reduce generation times by an order of magnitude.

This presents a strategic choice between two high-fidelity generation methods. The headless browser approach is inherently future-proof; as Stepan Mitkin updates drakonwidget.js with new features or format changes, our system will automatically adopt them by simply updating the library version. The direct .drn generation method, while significantly faster, depends on a documented schema. Although the documentation guarantees compatibility within a major version, a future major release of the DRAKON Editor could introduce schema changes that would require maintenance on our end.[5]

The optimal strategy is therefore a hybrid one:

- **Primary Method:** Implement the headless browser microservice. Its guarantee of forward compatibility makes it the most robust and lowest-maintenance option for day-to-day use.
- **Secondary Method:** Develop a module for direct .drn SQLite generation. This can be exposed as a "high-performance" mode for bulk processing or serve as a critical fallback if the headless browser service encounters issues.

Comparison of Diagram Generation Approaches

To clarify the trade-offs and justify the recommended hybrid strategy, the following table compares the three primary approaches to diagram generation.

Feature	**Manual JSON (Current)**	**Headless drakonwidget API**	**Direct .drn (SQLite) Generation**
Compatibility	Low (Brittle, format-dependent)	**Very High (Guaranteed by official library)**	High (Schema-dependent)
Performance	High	Low-Medium (Browser startup/IPC overhead)	**Very High (Native database I/O)**
Maintenance Effort	High (Requires tracking format changes)	**Very Low (Future-proof by design)**	Medium (Requires tracking schema changes)
Implementation	N/A (Exists)	Medium (Requires	Low-Medium (Requires SQLite

Effort		Node.js service)	logic)
Features Supported	Basic	**All (Auto-layout, advanced nodes, etc.)**	All (Requires manual implementation of features)
Recommendation	Deprecate	**Primary Method**	**High-Performance / Fallback Method**

Migration Plan

The transition from the current manual JSON generation to the proposed architecture can be executed with minimal disruption to the existing Python analysis pipeline.

1. **Develop the Node.js Service:** Create the microservice with its /generate endpoint and the Puppeteer-based logic for controlling drakonwidget.
2. **Define a Command Schema:** Establish a simple JSON schema for the "diagram commands" that the Python client will send. This schema should be high-level (e.g., {"command": "addNode", "type": "action", "content": "...", "parentId": "..."}).
3. **Refactor drakon_to_json.py:** Modify the existing Python code. Instead of building a nested dictionary representing the final JSON, it will now generate a list of these command objects.
4. **Integrate HTTP Client:** At the end of the analysis, drakon_to_json.py will serialize the command list to JSON and make a single POST request to the Node.js service. It will then receive the final, validated DRAKON JSON as the response.

This approach contains the majority of the new complexity within the dedicated microservice, allowing the Python code to remain focused on analysis while benefiting from a vastly more robust and compatible generation backend.

A Multi-Layered Approach to Code Semantic Analysis

To achieve the required leap in diagram fidelity—from 4-6 nodes to 20-30 for complex functions—the system must move beyond surface-level AST analysis and adopt a multi-layered approach that captures the deep semantic and control-flow structures of the code. This involves building a richer intermediate representation of the program and leveraging AI to interpret it.

Layer 1: Control Flow Graph (CFG) as the Foundation

The fundamental limitation of the current system is its reliance on a direct traversal of the AST. An AST represents the syntactic structure of the code, but not its execution flow. The ideal data structure for this purpose is a Control Flow Graph (CFG). A CFG models a program as a graph of basic blocks, where each block is a straight-line sequence of code with no jumps in or out. Edges between these blocks represent the jumps in the control flow (e.g., if-then-else branches, loop entries and exits, and function calls).

Mature open-source libraries are available to facilitate this transformation. For Python, libraries like py2cfg and staticfg can generate CFGs directly from source code.[6] For TypeScript and JavaScript, tools such as esgraph or styx can be applied to the AST produced by tree-sitter to build a corresponding CFG.[8]

The implementation will involve a new stage in the analysis pipeline:

1. Parse the source code into an AST using tree-sitter.
2. Feed the AST to a language-specific CFG generation library.
3. The core diagram generation logic will now traverse this CFG instead of the AST. Each basic block in the CFG will map to one or more DRAKON nodes, and the edges of the CFG will define the connections between these nodes.

This CFG-based approach inherently solves several of the stated problems. Complex conditional logic is no longer simplified to a single question node; instead, each branch of an if-elif-else chain becomes a distinct edge and path in the graph. Nested loops are represented naturally as cycles within the graph.

Most importantly, CFGs provide a robust framework for visualizing error handling. A try statement is represented as a basic block with multiple potential exit edges: a "normal" edge to the code following the try-catch structure, and one or more "exceptional" edges that lead directly to the corresponding catch or finally blocks.[9] By traversing these exceptional edges, the system can generate a complete and accurate error-handling.json diagram, faithfully representing all possible error propagation paths.

Layer 2: Data Flow Analysis (DFA) for Variable Intelligence

While the CFG maps out *where* the program's execution can go, it doesn't describe what the program's *data is doing* along those paths. To capture state transitions and variable lifecycles, we must introduce Data Flow Analysis (DFA). DFA is a technique used to gather information about the possible set of values that variables can hold at various points in a program.[12]

Powerful static analysis engines like GitHub's CodeQL provide sophisticated DFA capabilities for both Python and TypeScript.[13] CodeQL can model the program as a database and run queries to track the flow of data from "sources" (where a value is defined or introduced) to "sinks" (where it is used or modified). This is often framed as "taint tracking," where a variable is "tainted" at its source, and its influence is tracked throughout the program.[15]

This capability directly addresses the goal of visualizing state transitions. For the Python process_workflow example, we can configure a DFA to:

1. Identify current_state as a variable of interest.
2. Define any assignment to current_state as a "source" of a new state value.
3. Track the flow of this value through the CFG.
4. Whenever an assignment like current_state = 'PROCESS' is encountered, the system can insert a DRAKON action node explicitly labeled "Transition to PROCESS state".

The combination of CFG and DFA provides a rich, semantic view of the program. The CFG provides the structural backbone of the diagram (the paths), while the DFA provides the semantic content that annotates those paths (the data transformations and state changes). This integrated approach is the key to evolving the diagrams from a syntactic reflection of the code to a semantic model of its logic.

Layer 3: Advanced Control Flow Visualization Patterns

Modern programming languages include constructs that do not map cleanly to traditional flowcharts. The enhanced system must define specific DRAKON patterns to represent these constructs accurately.

Async/Await and Promise Chains

Asynchronous operations using async/await and Promise chains are ubiquitous in TypeScript. A simple linear representation is incorrect, as it fails to capture the non-blocking nature and the separate success/failure paths. Academic research has formalized the concept of a **Promise Graph** to model these interactions, where nodes represent promises and edges represent causal fulfillment or rejection relationships.[17]

This concept can be mapped to a DRAKON pattern:

- An await expression or a .then() call is represented by a unique "Asynchronous Action" node.
- This node has two distinct exit paths, adhering to DRAKON's "happy path" convention:
    1. **Success Path (one):** A vertical exit on the main "skewer" that represents the successful resolution of the promise. Control flows from here to the code that executes after the await.
    2. **Failure Path (two):** A horizontal exit to the right that represents the rejection of the promise. This path leads directly to the corresponding .catch() block or the function's exception handler.[18]

This pattern visually segregates synchronous and asynchronous control flow and makes error handling paths for asynchronous operations explicit.

Event-Driven Patterns and State Machines

The Python process_workflow sample is a classic example of a finite state machine. Such patterns are best visualized using conventions from UML State Diagrams, which define states, transitions, events, and guards.[19]

The system can be taught to recognize this pattern (a central loop controlled by a state variable) and apply a specialized DRAKON visualization strategy:

- Each elif current_state == 'X': block is encapsulated in a distinct, visually identifiable "State Node".
- The code within that block represents the actions performed while in that state.
- Assignments that modify the state variable (e.g., current_state = 'Y') are rendered as explicit transition lines connecting the current state node to the next.
- Conditional logic within a state that determines the next state (e.g., if result.success:) is represented as a guard condition on the transition line.

This transforms a flat, confusing loop into a clear and intuitive state transition diagram, vastly improving comprehension.

Layer 4: AI-Powered Intelligent Pattern Recognition

To automate the detection of these patterns and to further enrich the diagrams with semantic meaning, a hybrid AI architecture is proposed. This architecture combines the structural analysis capabilities of Graph Neural Networks with the semantic understanding of Transformer models.

Proposed AI Architecture: A GNN-Transformer Hybrid

1. **Graph Neural Networks (GNNs) for Structural Analysis:** GNNs are a class of neural networks specifically designed to operate on graph-structured data, making them perfectly suited for analyzing the CFGs and DFAs generated in our pipeline.[21] We will train a GNN on a labeled dataset of CFGs from the project's codebase. The GNN's task will be node and subgraph classification. It will learn to recognize the characteristic topological "shapes" of various programming constructs, such as:
    - **Design Patterns:** Identifying the graph structure of a State Machine, an Observer pattern, or a Command pattern.
    - **Anti-Patterns:** Detecting deeply nested conditional logic or "callback hell" in asynchronous code.
    - **Code Smells:** Flagging overly complex functions or redundant logical blocks.
2. **Transformer Models (CodeT5) for Semantic Analysis:** While GNNs can identify *what* a code structure is, they cannot interpret *what it means*. For this, we turn to Transformer models. CodeT5 is an ideal choice as its encoder-decoder architecture is adept at both code understanding and text generation tasks.[23] Once the GNN identifies a structurally significant subgraph (e.g., a complex business logic calculation), the source code for that subgraph is fed to CodeT5 to perform high-level semantic tasks, such as generating concise, human-readable summaries for node labels or inferring business intent from function and variable names.

The synergy between these two models is the core of the AI engine. The GNN acts as an efficient, structure-aware "focusing mechanism," identifying the most salient parts of the code graph. The Transformer then performs a deep semantic "dive" on these pre-selected, high-value regions. This is far more efficient and effective than applying a computationally expensive LLM to an entire source file indiscriminately. It allows the system to allocate its most powerful analytical resources to the code sections that need them most.

AI Model Architecture Comparison

The following table justifies the selection of a hybrid architecture by highlighting the complementary strengths of each model type for the required analysis tasks.

Capability	**Graph Neural Network (GNN)**	**Transformer (e.g., CodeT5)**	**Hybrid (GNN + Transformer)**
Primary Input	Graph (CFG, DFA)	Text (Source Code, Comments)	Graph + Text
Core Strength	**Structural/Topological Pattern Recognition**	**Semantic Understanding & Text Generation**	**Synergistic Combination**
Ideal Use Cases	Design Pattern Detection, Anti-Pattern Finding, Code Clustering	Code Summarization, Intent Extraction, Comment Analysis	Find pattern with GNN, summarize with Transformer
Code Understanding	Relational (How code components connect)	Semantic (What code components mean)	**Holistic (Structure + Meaning)**
Implementation	Requires graph representation of code	Can operate on raw code text	Requires integrated multi-stage pipeline
Recommendation	Use for structural analysis	Use for semantic enrichment	**Adopt as primary AI architecture**

Proposed System Architecture

The enhanced system is designed as a distributed architecture composed of two primary services: a Python backend for code analysis and a Node.js microservice for diagram generation. This separation of concerns ensures modularity, scalability, and maintainability.

System Diagram

The following diagram illustrates the flow of data and control through the proposed system.

Фрагмент коду

```
graph TD
    subgraph Code Analysis
        A --> B{Tree-sitter Parser};
        B --> C;
        C --> D{CFG/DFA Generator};
        D -- CFG/DFA Graph --> E{AI Enrichment Engine};
        E -- Enriched Graph --> F{Diagram Logic Builder};
        F -- Diagram Commands --> G;
    end

    subgraph Diagram Generation [Node.js Microservice]
        H[API Endpoint /generate] --> I{Puppeteer Controller};
        I --> J[Headless Chrome Instance];
        J -- Loads --> K;
        I -- Executes JS --> K;
        K -- Exports --> L;
        L --> H;
    end

    G -- POST Request --> H;
    H -- JSON Response --> G;
    G --> M;

    subgraph AI Enrichment Engine
        E_CFG --> E_GNN;
        E_GNN -- Identifies Subgraph --> E_Selector;
        E_Selector -- Code Snippet --> E_LLM;
        E_LLM -- Node Labels/Intent --> E_Merger;
        E_GNN -- Pattern Labels --> E_Merger;
        E_Merger --> E_Out[Enriched Graph];
    end

    E --- E_CFG;
    E_Out --- F;
```

Component Breakdown

Code Analysis (Python Backend)

This service is responsible for all aspects of source code parsing, analysis, and semantic enrichment.

- **Tree-sitter Parser:** This existing component remains unchanged. It provides the initial, high-fidelity AST from Python, TypeScript, and Bash source files.
- **CFG/DFA Generator:** This new component is the core of the deep analysis engine. It consumes the AST and uses libraries like py2cfg for Python or styx for TypeScript to construct a detailed Control Flow Graph.[6] It can also invoke the CodeQL engine to perform Data Flow Analysis and annotate the graph with variable lifecycle information.[13]
- **AI Enrichment Engine:** This module contains the hybrid GNN-Transformer models. It takes the raw CFG/DFA, passes it to the GNN to identify and label known structural patterns, and then uses the Transformer to generate semantic summaries for key code blocks. The output is an "enriched graph" annotated with both structural and semantic metadata.
- **Diagram Logic Builder:** This component traverses the final enriched graph. Instead of directly creating JSON, it translates the graph's structure and annotations into a sequence of high-level, abstract commands (e.g., create_action_node, connect_nodes_with_condition). This decouples the analysis logic from the specific API of the generation service.
- **HTTP Client:** A simple client (e.g., using the requests library) that serializes the command sequence and sends it via a POST request to the diagram generation service.

Diagram Generation (Node.js Microservice)

This service has one responsibility: to correctly and reliably generate valid DRAKON JSON using the official drakonwidget.js library.

- **API Endpoint:** A lightweight web server (e.g., using Express.js) that exposes a single /generate endpoint to accept the diagram command sequence from the Python backend.
- **Puppeteer Controller:** The orchestrator for the headless browser. It manages a pool of browser instances, receives incoming requests, injects the appropriate JavaScript payload to execute the diagram commands via the drakonwidget API, and extracts the resulting JSON output.
- **Headless Chrome Instance:** A sandboxed Chrome browser instance, running without a GUI. This provides the complete, native environment required by drakonwidget.js, including the DOM, window object, and Canvas rendering engine, ensuring flawless execution of the library's internal logic.

Intelligent Diagram Generation and Readability

A diagram that is exhaustively detailed can easily become unreadable. The final layer of intelligence in the system focuses on managing this trade-off, providing users with control over detail levels and generating node labels that convey intent rather than literal code.

Complexity-Driven Granularity Control

The key to balancing detail and readability is to move away from generating a single, static diagram and instead enable multiple, hierarchical levels of detail. The decision of what to expand or collapse will be driven by a data-driven metric that reflects human comprehension.

While Cyclomatic Complexity measures the number of testable paths in a function, **Cognitive Complexity** is a more modern and suitable metric for this purpose.[25] Developed by SonarSource, Cognitive Complexity measures the mental effort required for a human to understand a piece of code. It specifically penalizes constructs that break the linear flow of reading, such as nesting of loops and conditionals, and complex logical operators (&&, ||).[26] This aligns perfectly with the goal of managing visual clutter in a diagram.

The proposed algorithm for granularity control is as follows:

1. **Calculate Complexity:** For each function and for major sub-blocks within it (like loops or

complex conditionals), calculate the Cognitive Complexity score.
2. **Establish Thresholds:** Define a set of configurable thresholds, for example: Low Complexity (< 10), Medium Complexity (10-25), and High Complexity (> 25).
3. **User-Configurable Detail Level:** The system's CLI will accept a user-defined detail level, for instance, on a scale of 1 to 5.
4. **Map Complexity to Visualization:** The system will use a mapping algorithm to determine how to render code blocks based on their complexity and the user's chosen detail level.
   - **Detail Level 1 (Overview):** For a function with high overall complexity, this view would collapse all loops into single "Loop" nodes and group complex conditional chains into a single "Business Logic" node. The goal is to show the high-level structure only.
   - **Detail Level 3 (Default):** This view would expand most logic but would automatically identify self-contained blocks with high Cognitive Complexity (e.g., a complex data transformation algorithm) and represent them as a single, expandable "sub-diagram" node. This keeps the main diagram clean while allowing users to drill down.
   - **Detail Level 5 (Exhaustive):** This view renders every statement and branch as a distinct DRAKON node, providing a complete, fine-grained visualization for deep analysis.

This approach transforms the diagram from a static artifact into an interactive exploration tool, empowering developers to understand complex code at the level of abstraction that is most appropriate for their current task.

LLM-Powered Context-Aware Node Generation

To make diagrams truly useful, their node labels must communicate the *intent* of the code, not just its literal syntax. A node labeled processPermissions(user, level) is less informative than one labeled "Verify user has admin privileges". Large Language Models (LLMs) fine-tuned for code, such as CodeT5, are exceptionally capable at this kind of summarization and intent extraction task.[29]

The node labeling algorithm will be integrated into the analysis pipeline:

1. **Isolate Code Blocks:** For each basic block in the CFG that will become a DRAKON node, the corresponding source code snippet is extracted.
2. **Prompt Engineering:** The code snippet is embedded within a carefully crafted prompt that instructs the LLM on the desired output format.
   - **For Action Nodes:** "Summarize the following TypeScript code block into a short, imperative phrase suitable for a flowchart action node. Focus on the business logic or primary action being performed. Code: [code snippet]"

    - **For Question Nodes:** "Summarize the following conditional expression as a concise question suitable for a flowchart decision node. Code: [conditional expression]"
3. **LLM Inference:** The prompt is sent to the CodeT5 model. The model's generated text becomes the content of the DRAKON node.
4. **Contextual Awareness:** This process naturally leverages the full context available in the code. The LLM will consider variable names (parsing camelCase or snake_case), comments, and docstrings associated with the code block to infer its purpose.[31]
5. **Multi-Language Support:** By simply modifying the prompt (e.g., "Summarize the following... in Ukrainian"), the system can be extended to generate diagram labels in multiple languages.

This LLM-based approach ensures that the generated diagrams are not just structurally accurate but also semantically rich and immediately understandable to developers, project managers, and other stakeholders.

Implementation Plan and Risk Analysis

A phased, iterative approach is recommended for implementing these enhancements. This strategy allows for the delivery of incremental value, facilitates feedback loops, and manages technical complexity by tackling the most critical dependencies first.

Phased Rollout Plan

- **Phase 1: The Generation Backbone (Estimated Effort: 2-3 weeks)**
    - **Tasks:**
        - Develop the Node.js microservice using Express.js and Puppeteer.
        - Create the minimal HTML host page for drakonwidget.js.
        - Implement the API wrapper to expose high-level diagram construction functions.
        - Refactor code_to_drakon.py to replace manual JSON building with an HTTP client that calls the new service.
    - **Goal:** Achieve 100% valid and compatible diagram generation. The output will still be based on the simple AST analysis, but the foundational generation mechanism will be robust, maintainable, and future-proof.
- **Phase 2: Deep Control Flow Analysis (Estimated Effort: 3-4 weeks)**
    - **Tasks:**
        - Integrate py2cfg for Python and a suitable CFG library (e.g., styx) for TypeScript.
        - Rewrite the Diagram Logic Builder to traverse the CFG instead of the AST.

        - Implement the logic to correctly map all CFG structures (branches, loops, exception paths) to DRAKON elements.
    - **Goal:** Achieve the target node density and control flow fidelity. Diagrams for complex functions like the sample handleUserRequest should now contain 15-20+ nodes, accurately representing all logical paths.
- **Phase 3: AI Enrichment and Semantics (Estimated Effort: 4-6 weeks, can run in parallel with Phase 2)**
    - **Tasks:**
        - Set up, train, and fine-tune a CodeT5 model on project-specific code for the summarization task.
        - Integrate the CodeT5 model into the analysis pipeline to generate node labels.
        - (Stretch Goal) Build a dataset of CFGs labeled with design patterns. Train a GNN model (e.g., using PyTorch Geometric) to detect 3-5 key patterns.
    - **Goal:** Generate diagrams with human-readable, intent-driven node labels. Achieve >80% similarity between AI-generated descriptions and human intent.
- **Phase 4: Advanced Visualization and User Experience (Estimated Effort: 2-3 weeks)**
    - **Tasks:**
        - Implement the Cognitive Complexity calculation algorithm.
        - Integrate the complexity-driven logic for automatic node collapsing and expansion.
        - Implement the specialized DRAKON patterns for async/await (Promise Graphs) and state machines.
        - Expose the "detail level" as a configurable parameter in the system's CLI.
    - **Goal:** Deliver hierarchical, interactive diagrams and provide accurate, intuitive visualizations for modern programming constructs.

Risk Analysis and Mitigation Strategies

- **Risk 1: drakonwidget Internal API Changes (Probability: Medium, Impact: High)**
    - **Description:** The headless browser architecture depends on the internal, undocumented API of drakonwidget.js. A future update to the library could introduce breaking changes without warning.
    - **Mitigation:**
        1. **Anti-Corruption Layer:** The Node.js microservice is designed as an anti-corruption layer, isolating the rest of the system from these potential changes.
        2. **Integration Testing:** A comprehensive suite of integration tests will be developed. These tests will generate a set of canonical diagrams and perform snapshot testing on the resulting JSON output. This test suite will be run against new versions of drakonwidget.js before deployment, providing immediate

    detection of any breaking changes.
  3. **Fallback Strategy:** The direct .drn SQLite generation capability will be maintained as a robust fallback, ensuring business continuity even if the primary generation method is temporarily broken.
- **Risk 2: Performance of Headless Browser (Probability: High, Impact: Medium)**
  - **Description:** The overhead of starting a new headless browser instance for each diagram generation could be significant, potentially violating the < 5 seconds performance requirement for individual functions and being prohibitive for large-scale batch processing.
  - **Mitigation:**
    1. **Instance Pooling:** The Node.js service will implement a persistent pool of "warm" headless browser instances. This eliminates the costly startup and teardown process for each request, drastically reducing latency.
    2. **High-Performance Mode:** For bulk processing of entire codebases, the system will provide a CLI flag to use the direct .drn SQLite generation method, which avoids browser overhead entirely and offers superior throughput.
- **Risk 3: AI Model Accuracy and Hallucination (Probability: Medium, Impact: Medium)**
  - **Description:** The LLM used for node summarization may occasionally produce inaccurate, irrelevant, or nonsensical labels ("hallucinations"), degrading the quality of the diagrams.
  - **Mitigation:**
    1. **Few-Shot Prompting:** The prompts sent to the LLM will include several high-quality examples of code snippets and their desired summaries. This "few-shot learning" technique helps to guide the model's output and significantly improves accuracy and consistency.
    2. **Confidence Scoring and Fallback:** The system will analyze the LLM's output. If the output is malformed, too long, or has a low confidence score from the model, the system will automatically revert to a simpler, deterministic labeling strategy (e.g., using the function call name or the first line of the code block). This ensures a baseline level of quality for all nodes.

Appendix

A.1. Algorithm Pseudocode

Pseudocode: CFG Traversal for DRAKON Generation

```
function generateFromCFG(cfg, drakonBuilder):
 visited_blocks = new Set()
 worklist = new Queue([cfg.entry_block])

 while not worklist.isEmpty():
  current_block = worklist.dequeue()
  if current_block in visited_blocks:
   continue
  visited_blocks.add(current_block)

  parent_node_id = process_block_statements(current_block, drakonBuilder)

  for exit_link in current_block.exits:
   next_block = exit_link.target_block

   if exit_link.is_conditional():
    // Create a 'branch' node
    branch_node_id = drakonBuilder.addBranch(exit_link.condition, parent_node_id)

    // Connect 'yes' path
    connect_to_block(branch_node_id, 'one', next_block, drakonBuilder)

    // Connect 'no' path to the other exit
    else_block = find_else_block(current_block, exit_link)
    connect_to_block(branch_node_id, 'two', else_block, drakonBuilder)

    worklist.enqueue(else_block)
   else:
    // Unconditional link
    connect_to_block(parent_node_id, 'one', next_block, drakonBuilder)

   worklist.enqueue(next_block)

function process_block_statements(block, builder):
 // Process statements within the block, creating 'action' nodes
 //... returns the ID of the last node created for this block
```

Pseudocode: Cognitive Complexity to Detail Level Mapping

```
function shouldCollapseNode(code_block, user_detail_level):
 // user_detail_level is 1 (min detail) to 5 (max detail)

 cognitive_complexity = calculateCognitiveComplexity(code_block)

 // Define thresholds
 LOW_COMPLEXITY = 10
 HIGH_COMPLEXITY = 25

 // Higher detail level means lower collapse threshold
 collapse_threshold = HIGH_COMPLEXITY - (user_detail_level * 4)

 if cognitive_complexity > collapse_threshold:
   return true
 else:
   return false
```

A.2. Tool and Library Recommendations

Category	Tool/Library	Language	Rationale
AST Parsing	tree-sitter	Python/JS	High-performance, incremental parsing for multiple languages. Already

			in use.
CFG Generation	py2cfg	Python	Generates CFGs from Python 3 source, visualizable with Graphviz.[6]
	styx	TypeScript	Creates CFGs from ESTree-compliant ASTs, suitable for TS/JS.[8]
DFA / Taint Tracking	CodeQL	Python/TS	Industry-leading static analysis engine for deep data flow and security analysis.[13]
Headless Browser	Puppeteer	Node.js	Provides robust, programmatic control over headless Chrome for server-side automation.[3]
Web Service	Express.js	Node.js	Mature, simple, and fast framework for building the Node.js microservice API.
LLM for Code	CodeT5 (Hugging Face)	Python	Open-source encoder-decoder model excelling at code summarization and generation.[23]
GNN Framework	PyTorch Geometric	Python	Powerful and flexible library for implementing Graph Neural Networks on

			custom graph data.

A.3..drn SQLite Schema Reference

The following is a summary of the key tables and fields in the .drn file format, as documented for the DRAKON Editor. This schema is critical for implementing the direct, high-performance generation mode.[5]

- **diagrams table:** Contains metadata for each diagram.
  - diagram_id (integer, PK): Unique ID for the diagram.
  - name (text, unique): The human-readable name of the diagram.
  - zoom (double): The default zoom level.
- **items table:** Contains the definitions for every icon and connector on a diagram.
  - item_id (integer, PK): Unique ID for the item.
  - diagram_id (integer, FK): References the diagram this item belongs to.
  - type (text): The type of the item (e.g., action, branch, end, header).
  - text (text): The primary content/label for the icon.
  - text2 (text): Secondary text field for certain icon types.
  - one (integer): The item_id of the item connected to the primary (Yes/vertical) exit.
  - two (integer): The item_id of the item connected to the secondary (No/right) exit.
  - three (integer): The item_id for the third exit (used in case statements).
  - x, y, w, h (integer): Positional and dimensional data for the item on the canvas.

Джерела

1. stepan-mitkin/drakonwidget: A JavaScript widget for ... - GitHub, доступ отримано жовтня 19, 2025, https://github.com/stepan-mitkin/drakonwidget
2. DRAKON Editor, доступ отримано жовтня 19, 2025, https://drakon-editor.sourceforge.net/
3. The Best Node.js Headless Browsers for Web Scraping, доступ отримано жовтня 19, 2025, https://scrapeops.io/nodejs-web-scraping-playbook/best-nodejs-headless-browsers/
4. Headless Browser in NodeJS with Puppeteer [2025] - ZenRows, доступ отримано жовтня 19, 2025, https://www.zenrows.com/blog/headless-browser-nodejs
5. DRAKON Editor 1.22 .drn file format, доступ отримано жовтня 19, 2025, https://drakon-editor.sourceforge.net/file_format.html
6. py2cfg - PyPI, доступ отримано жовтня 19, 2025, https://pypi.org/project/py2cfg/
7. coetaur0/staticfg: Python3 control flow graph generator - GitHub, доступ отримано жовтня 19, 2025, https://github.com/coetaur0/staticfg

8. control flow graph - npm search, доступ отримано жовтня 19, 2025, https://www.npmjs.com/search?q=control%20flow%20graph
9. The ultimate guide to Python exception handling - Honeybadger Developer Blog, доступ отримано жовтня 19, 2025, https://www.honeybadger.io/blog/a-guide-to-exception-handling-in-python/
10. Exception & Error Handling in Python | Tutorial by DataCamp, доступ отримано жовтня 19, 2025, https://www.datacamp.com/tutorial/exception-handling-python
11. try...catch - JavaScript - MDN - Mozilla, доступ отримано жовтня 19, 2025, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try...catch
12. Data-flow analysis - Wikipedia, доступ отримано жовтня 19, 2025, https://en.wikipedia.org/wiki/Data-flow_analysis
13. Analyzing data flow in Python - CodeQL - GitHub, доступ отримано жовтня 19, 2025, https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-python/
14. Analyzing data flow in JavaScript and TypeScript - CodeQL - GitHub, доступ отримано жовтня 19, 2025, https://codeql.github.com/docs/codeql-language-guides/analyzing-data-flow-in-javascript-and-typescript/
15. About data flow analysis - CodeQL - GitHub, доступ отримано жовтня 19, 2025, https://codeql.github.com/docs/writing-codeql-queries/about-data-flow-analysis/
16. TaintTracking - CodeQL - GitHub, доступ отримано жовтня 19, 2025, https://codeql.github.com/codeql-standard-libraries/javascript/semmle/javascript/dataflow/TaintTracking.qll/module.TaintTracking.html
17. A model for reasoning about JavaScript promises | Request PDF - ResearchGate, доступ отримано жовтня 19, 2025, https://www.researchgate.net/publication/320391516_A_model_for_reasoning_about_JavaScript_promises
18. Using promises - JavaScript | MDN, доступ отримано жовтня 19, 2025, https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Using_promises
19. A simple guide to drawing your first state diagram (with examples) - Nulab, доступ отримано жовтня 19, 2025, https://nulab.com/learn/software-development/a-simple-guide-to-drawing-your-first-state-diagram-with-examples/
20. State Machine Diagram Tutorial - Lucidchart, доступ отримано жовтня 19, 2025, https://www.lucidchart.com/pages/uml-state-machine-diagram
21. What is a GNN (graph neural network)? - IBM, доступ отримано жовтня 19, 2025, https://www.ibm.com/think/topics/graph-neural-network
22. Graph Neural Networks on Program Analysis - Miltos Allamanis ·, доступ отримано жовтня 19, 2025, https://miltos.allamanis.com/publicationfiles/allamanis2021graph/allamanis2021graph.pdf
23. What Is CodeT5? Key Features & What Makes It Unique - Milestone AI, доступ

отримано жовтня 19, 2025, https://mstone.ai/tools-wizard/code-t5/
24. Cracking the Code LLMs | Towards Data Science, доступ отримано жовтня 19, 2025, https://towardsdatascience.com/cracking-the-code-llms-354505c53295/
25. Cyclomatic Complexity vs Cognitive Complexity: Key Differences Explained - Graph AI, доступ отримано жовтня 19, 2025, https://www.graphapp.ai/blog/cyclomatic-complexity-vs-cognitive-complexity-key-differences-explained
26. (PDF) Comparative Analysis between Cognitive Complexity and Cyclomatic Complexity in Software Development - ResearchGate, доступ отримано жовтня 19, 2025, https://www.researchgate.net/publication/389175932_Comparative_Analysis_between_Cognitive_Complexity_and_Cyclomatic_Complexity_in_Software_Development
27. What is the difference between Cyclomatic Complexity and Cognitive Complexity ? | by Gilles Fabre, доступ отримано жовтня 19, 2025, https://gilles-fabre.medium.com/what-is-the-difference-between-cyclomatic-complexity-and-cognitive-complexity-a87cef0e2851
28. Cognitive Complexity Vs Cyclomatic Complexity - An Example With C# - C# Corner, доступ отримано жовтня 19, 2025, https://www.c-sharpcorner.com/blogs/cognitive-complexity-vs-cyclomatic-complexity-an-example-with-c-sharp
29. Few-shot training LLMs for project-specific code-summarization - ResearchGate, доступ отримано жовтня 19, 2025, https://www.researchgate.net/publication/366918308_Few-shot_training_LLMs_for_project-specific_code-summarization
30. SpecRover: Code Intent Extraction via LLMs - Abhik Roychoudhury, доступ отримано жовтня 19, 2025, https://abhikrc.com/pdf/ICSE25.pdf
31. PROCONSUL: Project Context for Code Summarization with LLMs - ACL Anthology, доступ отримано жовтня 19, 2025, https://aclanthology.org/2024.emnlp-industry.65.pdf