CSCE- 689: PROGRAMMING IN LLMs - Project Report

Name - Navya Priya Nandimandalam
UIN- 136004813

RepoSpeak: Code Analysis Through Voice & Vision

Abstract

Developers often struggle to understand unfamiliar codebases, especially when documentation is incomplete or outdated. This project presents RepoSpeak, a tool that combines traditional code analysis with modern AI to make Python repositories easier to understand. Unlike conventional tools that only extract syntactic information, RepoSpeak uses a hybrid approach: Abstract Syntax Tree (AST) parsing handles structural analysis while Claude Sonnet 4.5 provides semantic understanding to explain what the code does and why it matters. The system generates human-readable and technical summaries at multiple levels, creates audio narrations, produces architectural diagrams, and detects dead code with contextual reasoning to distinguish genuine dead code from false positives. We evaluated RepoSpeak using an LLM-as-a-Judge methodology with Claude Opus 4.5 across six Python repositories of varying complexity. Results demonstrate that the hybrid approach achieves high precision by combining AST efficiency with LLM nuanced interpretation, significantly reducing false positives while producing accurate documentation. RepoSpeak shows how AI can enhance traditional code analysis to help developers onboard faster and maintain cleaner codebases.

Introduction

- **Problem Statement**

  Software repositories are growing increasingly complex, making it challenging for developers to understand, maintain, and onboard to new codebases. Studies show that developers spend up to 60% of their time reading and understanding existing code rather than writing new features. This challenge is particularly acute in Python repositories, which are heavily used across data science, web development, and automation, yet frequently lack comprehensive documentation due to rapid development cycles. Traditional documentation approaches like README files, inline comments, and manually written guides, often become outdated as code evolves, creating gaps between what the documentation says and what the code actually does.

  Developers face three critical challenges when approaching unfamiliar codebases:

  - **Grasping the high-level architecture and purpose of a repository**: Traditional static analysis tools like Pylint or Radon provide metrics about code complexity and style violations but fail to convey the semantic intent or the "story" of the code. These tools can identify what the code is structurally, but not why it exists or how components work together to solve problems.

  - **Identifying and understanding technical debt, particularly dead code:** Tools like Vulture detect potentially unused code through static analysis but suffer from high false positive rates. They cannot distinguish between genuinely dead code and legitimate use cases, for example, functions exported as public APIs, methods called indirectly through decorators or callbacks, or classes implementing protocol interfaces. Manual inspection of hundreds of flagged items becomes impractical, yet blindly deleting code risks breaking functionality.

  - **Consuming documentation in formats suited to diverse learning preferences**: Some developers prefer visual diagrams showing module relationships, others benefit from audio explanations, and many need both high-level overviews for stakeholders and detailed technical summaries for implementation. Existing tools provide only textual output or basic dependency graphs, failing to support different modalities and learning styles.

- **Motivation**

  Recent advances in Large Language Models (LLMs) present an opportunity to transform how developers understand code. Models like Claude Sonnet 4.5, GPT-4, and others demonstrate remarkable abilities to comprehend programming languages, explain complex logic in natural language, and reason about software architecture at semantic levels that go beyond syntactic pattern matching. However, using LLMs alone for code analysis has significant limitations, for example, they can hallucinate incorrect information when not grounded in actual code structure, lack the systematic rigor and reliability of deterministic parsing, and may miss patterns that are obvious to traditional static analysis techniques.

  This motivates a hybrid approach that combines the complementary strengths of both techniques. Abstract Syntax Tree (AST) parsing provides reliable, deterministic extraction of code structure like the function signatures with type annotations, class hierarchies with inheritance relationships, import dependencies, and complete call graphs. This structural foundation is fast as compared to parsing thousands of lines per second, accurate with no hallucinations, and scales efficiently to large repositories. LLMs add the critical semantic layer, that is, they can explain why code exists based on naming patterns and logic flow, what problem it solves by understanding domain context, and how components relate in ways that transcend simple structural connections.

  Especially for dead code detection, AST parsing efficiently identifies candidates: unreferenced functions, uninstantiated classes, unread variables, and unreachable code blocks, but cannot determine intent or context. Our implementation uses Claude Sonnet 4.5 to analyze each candidate with contextual awareness, checking whether it's part of a public API, implements framework protocols, serves as a callback, or has configuration purposes. The LLM provides three-way classification (dead code, false positive, uncertain) with confidence scores, detailed reasoning, and actionable recommendations.

- **Objectives**

  This project develops RepoSpeak, an intelligent, multi-modal code analysis system that transforms Python repositories into comprehensive, accessible documentation. The system implements four primary objectives:

  - **Hierarchical Code Summarization**: Generate function-level, module-level, and repository-level summaries using AST parsing combined with LLM analysis. Provide both human-readable and technical descriptions at each level. For large functions, use chunking strategies to maintain context across the entire codebase.

  - **Intelligent Architecture Visualization**: Create high-level architectural diagrams using Mermaid.js that group related modules by logical purpose. Use LLM analysis along with AST results to identify semantic categories and visualize module relationships with color-coded architectural layers.

  - **Audio Documentation**: Produce AI-generated audio narrations that explain the repository in conversational terms using analogies, making documentation accessible to auditory learners and enabling hands-free code understanding.

  - **Hybrid Dead Code Detection**: Identify potentially unused code across various categories using AST analysis. Validate each candidate with LLM reasoning to distinguish genuine dead code from false positives, providing confidence scores, detailed explanations, and actionable recommendations.

- **Key Contributions**

  This project makes several contributions to automate code comprehension:

  - We demonstrate a practical hybrid architecture that combines AST parsing with LLM reasoning for multi-modal documentation generation and dead code detection in a unified system. Our implementation spans over 4,000 lines of Python code across 10 core modules, showing that hybrid approaches can scale to real-world repositories.

  - We develop context-aware prompting strategies that enable LLMs to distinguish genuine dead code from false positives by considering project type (library vs. application), framework conventions, and architectural patterns. This context-driven approach significantly reduces false positives compared to pure static analysis tools.

  - We present an LLM-as-a-Judge evaluation methodology using Claude Opus 4.5 to provide consistent, reproducible assessment of both multi-modal documentation quality and dead code detection accuracy across diverse repositories. This addresses the challenge of evaluating subjective aspects like clarity and usefulness while establishing ground truth for dead code.

  - We provide empirical evidence across six real-world Python repositories of varying complexity, from simple CLI applications to popular libraries like requests, rich, and click, demonstrating that the hybrid approach achieves high precision in dead code detection while producing documentation that scores at least 4 out of 5 on accuracy, completeness, and clarity.

Related Work

Our project builds on recent advances in LLM-based code analysis and hybrid approaches that combine static analysis with machine learning. We review prior work in two key areas and identify gaps that RepoSpeak addresses.

- **LLM-Based Code Documentation**

  RepoAgent by Luo et al. (2024 - https://arxiv.org/abs/2402.16667) introduced an LLM-powered open-source framework for automated repository-level code documentation generation. Unlike traditional tools that document individual functions or files in isolation, RepoAgent operates at the repository level, generating and maintaining documentation that captures cross-file dependencies and component interactions. The system leverages LLMs to proactively generate, maintain, and update code documentation, addressing the challenge of keeping documentation synchronized with evolving codebases. Their evaluation combined qualitative assessment of documentation quality and quantitative metrics, demonstrating that the approach excels in generating high-quality repository-level documentation across several open-source Python projects.

  However, RepoAgent has significant limitations. It produces only text-based documentation in a single format, requiring developers to read through potentially lengthy documents. The system does not provide audio narrations for hands-free learning or visual diagrams to quickly grasp architectural relationships. Additionally, RepoAgent focuses purely on documentation generation and does not address code quality issues like identifying dead code or technical debt, which are critical concerns for repository maintainability.

  Diggs et al. (2024 - https://arxiv.org/html/2411.14971v1) investigated using LLMs to generate documentation for legacy code in MUMPS and mainframe assembly languages. These archaic languages are still used in critical healthcare and financial systems but lack modern tooling and developer expertise. Their approach involves extracting code snippets and using a novel prompting strategy where existing comments are replaced with unique identifiers, the LLM generates comments in structured JSON format mapping identifiers to explanations, and a greedy chunking algorithm optimizes context window usage. They evaluated four LLMs (Claude 3.0 Sonnet, Llama 3, Mixtral, GPT-4 Turbo) by having domain experts rate generated comments on hallucination, readability, completeness, and usefulness using a four-point

scale. For MUMPS code, they found that LLM-generated comments achieved ratings comparable to human-written documentation, with GPT-4 scoring 9.1/10 for factual accuracy and producing more readable comments than original developers. However, assembly language comments performed poorly, and automated metrics showed no strong correlation with human quality judgments.

While this work demonstrates LLMs' ability to understand unfamiliar legacy programming paradigms, it focuses narrowly on inline comment generation for individual code snippets. The approach does not attempt repository-level understanding, architectural visualization, or identifying unused code. Furthermore, their evaluation highlights a critical challenge: while manual expert review provides accurate quality assessment, it is expensive and does not scale to large codebases. The authors attempted to validate automated metrics (ROUGE, BLEU, cosine similarity) against human judgments but found that none correlated strongly with perceived quality and the best-performing automated metric achieved only 0.34 correlation with human usefulness ratings. This reveals a fundamental gap in the ability to automatically evaluate documentation quality, making it difficult to assess LLM-generated documentation at scale.

- **Hybrid Static Analysis with LLMs**

  Li et al. (2024 - https://dl.acm.org/doi/10.1145/3649828) presented a framework synergizing static analysis with LLMs for detecting "Use Before Initialization" (UBI) bugs in the Linux kernel. UBI bugs occur when a variable is read before it has been assigned a value, potentially causing crashes or security vulnerabilities. Traditional static analysis can identify potential UBI instances by tracking variable assignments and reads along execution paths. However, these tools suffer from high false positive rates because they struggle with complex control flow, conditional assignments, and aliasing.

  Their hybrid approach addresses this by using static analysis to construct path constraints (conditions that must hold for a particular execution path to be taken), then feeding these constraints to an LLM along with code context. The LLM performs constraint-guided reasoning, determining whether the constraints are satisfiable and whether a UBI can actually occur under realistic program conditions. By combining static analysis's systematic path exploration with the LLM's ability to reason about complex logical conditions, they achieved significantly higher precision (fewer false positives) than traditional static analyzers while maintaining high recall.

  Li et al. (2025 - https://arxiv.org/abs/2405.17238) proposed IRIS, a neuro-symbolic approach for security vulnerability detection that combines LLMs with static analysis to address taint analysis problems. Traditional static analysis tools like CodeQL require manually specified taint specifications for identifying which functions introduce taint (sources), remove taint (sanitizers), or execute dangerous operations (sinks), which is time-consuming and requires expertise. IRIS automates this process by using LLMs to infer these specifications by analyzing function names, signatures, and implementations.

  Once specifications are inferred, traditional static analysis performs whole-repository data flow analysis to track taint propagation precisely and scalably. This division of labor leverages LLMs' semantic understanding of code purpose and static analysis's rigorous whole-program reasoning. Evaluation on CWE-Bench-Java showed IRIS detected 55 vulnerabilities compared to CodeQL's 27, while discovering 4 previously unknown vulnerabilities, demonstrating significant improvement over traditional static analysis.

- **How RepoSpeak Addresses These Gaps**

  RepoSpeak makes four key advances over prior work:

  - **Multi-Modal Documentation**: Unlike RepoAgent and Diggs et al., which generate only text, we produce documentation in multiple formats. Audio narrations enable hands-free learning. Mermaid.js diagrams provide visual overviews of architectural relationships and module groupings based on semantic purpose rather than directory structure. Human-readable summaries serve non-technical stakeholders, while technical summaries provide implementation details for developers. This multi-modal approach accommodates different learning preferences and use cases.

- **Integrated Dead Code Detection**: We extend hybrid analysis beyond bugs (Li et al.) to dead code detection. Our system identifies six categories of potentially unused code through AST analysis, then uses Claude Sonnet 4.5 to validate each candidate with rich contextual reasoning. The LLM considers whether code is part of a public API (exported in __init__.py), implements framework protocols (e.g., PyTorch's forward() method), serves as a callback, or has configuration purposes. This context-aware validation distinguishes genuine dead code from false positives, a critical improvement over tools like Vulture that rely purely on static call graphs.

- **LLM-as-a-Judge Evaluation**: We introduce an automated evaluation methodology using Claude Opus 4.5 to assess both documentation quality (accuracy, completeness, clarity, usefulness) and dead code detection accuracy (comparing Opus's independent analysis against Sonnet's classifications). This provides scalable, reproducible evaluation across diverse repositories, addressing the limitations of manual expert review (Diggs et al.) and the need for expensive ground-truth datasets (Li et al.).

  By integrating documentation generation with dead code analysis in a multi-modal, context-aware system, RepoSpeak provides a more comprehensive solution to repository understanding and maintenance than prior work.

Methodology

- **System Overview**: RepoSpeak implements a multi-stage pipeline that transforms Python repositories into multi-modal documentation through seven core components:

  - AST Parser - Extracts code structure, dependencies, and dead code candidates
  - Function Summarizer - Generates human-readable and technical function summaries
  - Module Summarizer - Aggregates function summaries into module(every python file) descriptions
  - Repository Summarizer - Creates repository-level overviews
  - Dead Code Analyzer - Validates AST-detected candidates with LLM reasoning
  - Module Grouper & Diagram Generator - Produces semantic architectural visualizations
  - Audio Generator - Converts summaries to speech narration

  The system follows a bottom-up aggregation strategy. First, the AST Parser analyzes all Python files, extracting structural information and identifying potential dead code through static analysis. This AST data feeds into the Function Summarizer, which uses Claude Sonnet 4.5 to generate dual summaries (human-readable and technical) for each significant function. Function summaries aggregate into module summaries, which further combine into a repository-level overview. In parallel, the Dead Code Analyzer validates AST-detected candidates by providing Claude Sonnet with rich context about each candidate's purpose and usage patterns, classifying each as genuine dead code, false positive, or uncertain. The Module Grouper uses Claude's semantic understanding of module summaries to identify logical architectural groups, which the Diagram Generator visualizes as color-coded Mermaid.js diagrams. Finally, the Audio Generator uses the repository summary as context to create conversational MP3 narration with analogies, making technical concepts accessible through audio. Throughout the pipeline, a Context Manager maintains all intermediate results in JSON format, enabling reproducibility and resumable analysis. The complete system architecture, showing the multi-stage pipeline and data flow, is illustrated in Figure 1 below.

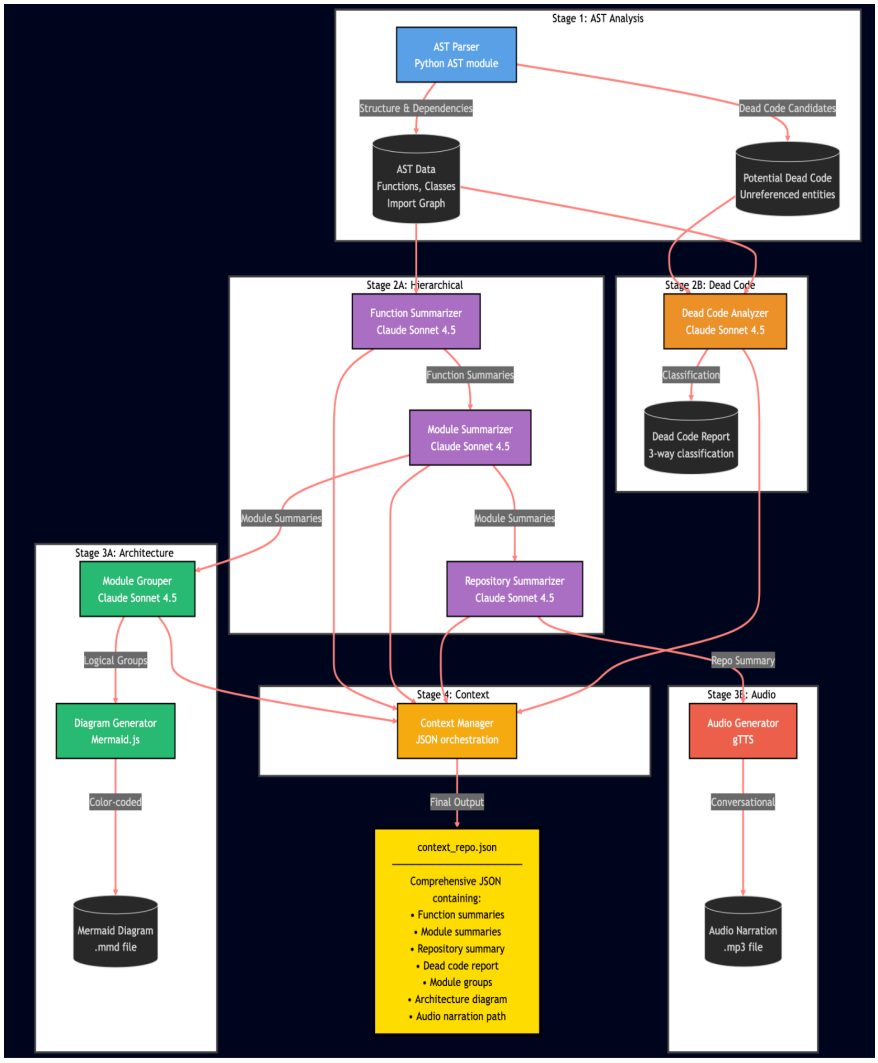


Figure 1: RepoSpeak system pipeline

- **AST Parser**: The AST Parser extracts structural information from Python repositories using Python's ast module without code execution, ensuring safe analysis of unfamiliar codebases without requiring runtime dependencies or risking malicious code execution. It captures function signatures with type hints and decorators, class hierarchies, import dependencies classified into standard library versus third-party packages, module-level variables, and entry points. The parser builds a cross-module dependency graph tracking all defined and called functions, defined and instantiated classes, enabling repository-wide dead code detection across file boundaries.

  The parser identifies six categories of dead code candidates. Unreferenced functions are those never called anywhere in the codebase, excluding special methods, decorator-driven functions (e.g., @app.route, @pytest.fixture), framework implicit methods (e.g., PyTorch's forward(), scikit-learn's fit()), and exported API symbols. Unused classes are never instantiated or inherited. Unused imports are symbols never referenced in code, excluding __init__.py exports. Unused global variables are written but never read. Unreachable code includes statements after return/raise, if False: blocks, and while False: loops. Suspicious patterns detect code smells like functions with many parameters (>6), very long functions (>100 lines), and functions called only once.

  To minimize false positives and avoid flagging legitimate public API elements, the parser implements export detection by analyzing __init__.py imports, __all__ declarations, and config files (conf.py, setup.py,

conftest.py), marking these symbols as public API and excluding them from dead code detection. This distinction is critical because libraries expose functions for external consumption that appear unused internally. Each candidate is tagged with needs_llm: true/false to optimize processing, for example, unused imports receive false as they are definitively unused based on static analysis alone and can be auto-flagged without LLM cost, while unreferenced functions, unused classes, and unreachable code receive true because they require semantic context from the LLM to distinguish reflection-based invocation, plugin architectures, and intentionally disabled features from genuine dead code.

- **Hierarchical Summarization Strategy**: RepoSpeak implements bottom-up aggregation across three levels to build comprehensive documentation that maintains consistency between repository overview and implementation details. The hierarchical approach avoids hallucinations by grounding each level of abstraction in concrete code analysis from the level below.

    - **Function Summarizer**: Generates dual summaries for each function, that is, human-readable explanations for developers unfamiliar with the codebase, and technical specifications preserving implementation details for maintenance, serving different audiences with appropriate levels of abstraction. The summarizer receives AST-extracted function data including the complete source code, function signature with type hints, decorators, existing docstring, line count, and the list of functions called within the function body. For functions exceeding 200 lines (approximately 12,000 characters), a chunking strategy splits code into 60-line segments with 15-line overlap to fit LLM context windows. The overlap ensures code spanning chunk boundaries remains interpretable. Each chunk receives context about the overall function signature, variables, and purpose, enabling the LLM to understand how the chunk fits into the larger function. Partial summaries from each chunk are then aggregated into a coherent function-level summary that captures the complete logic flow without losing details to context limitations. The workflow for summarizing small versus large functions is shown in Figure 2 below.

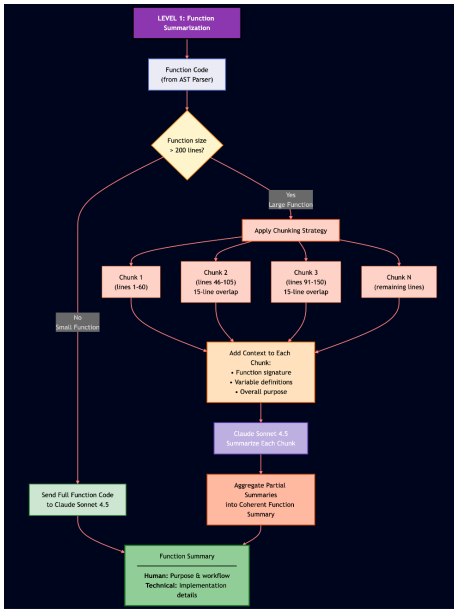


Figure 2: Function summarization workflow

    - **Module Summarizer**: Aggregates function summaries into module-level descriptions, where each module corresponds to a single Python file in the repository. The summarizer receives function summaries along with AST-extracted metadata including class definitions (names, methods, base classes), import statements (dependencies, aliases), and function metadata (signatures, decorators, line counts). It processes up to 30 function summaries per API call to produce dual summaries identifying the module's architectural role, key exported functions and classes, design patterns employed, and dependencies on other modules. For large modules

exceeding 30 functions, the summarizer applies a chunking strategy: it divides function summaries into groups of 30, generates intermediate summaries for each group describing their collective purpose, then combines these group summaries into a final module-level summary that captures the complete module functionality. This chunking approach enables summarization of arbitrarily large Python files while maintaining coherence and avoiding context window limitations. The module-level perspective reveals cohesion and coupling patterns not visible at the function level, for example, a file containing functions for token parsing, syntax validation, and AST construction would be summarized as a "Syntax Analysis Module" rather than just listing individual functions. This middle layer of abstraction essentially helps developers understand module boundaries and identify refactoring opportunities without getting lost in low-level function details. Figure 3 presents the module summarization process that aggregates function summaries.

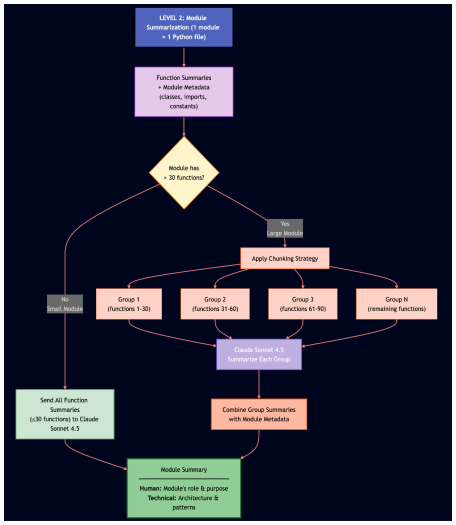


Figure 3: Module summarization workflow

- **Repository Summarizer**: Synthesizes up to 40 module summaries along with AST-derived repository statistics (total modules, total functions, total classes) into a repository-level overview explaining what the project does, how it's structured, what technologies it uses. When repositories contain more than 40 modules, the summarizer uses a two-stage approach: first grouping modules by directory (like src/core, src/api, tests) and summarizing each group's purpose, then combining these group summaries into a cohesive repository overview. The 40-module limit balances providing comprehensive context to the LLM while staying within token budgets for a single API call and this hierarchical strategy handles large codebases while maintaining a clear architectural picture. The summarizer also distinguishes between library repositories, which need documentation emphasizing public APIs and extensibility and application repositories, which need documentation emphasizing user workflows and entry points. The repository summary serves as the foundation for audio generation and gives newcomers a mental map of the project before they dive into specific modules or functions. Because the summary builds on actual module summaries rather than analyzing raw code directly, it reflects the real implemented architecture instead of hallucinating features or components that don't exist. The process for generating a repository-level summary is shown in Figure 4 below.

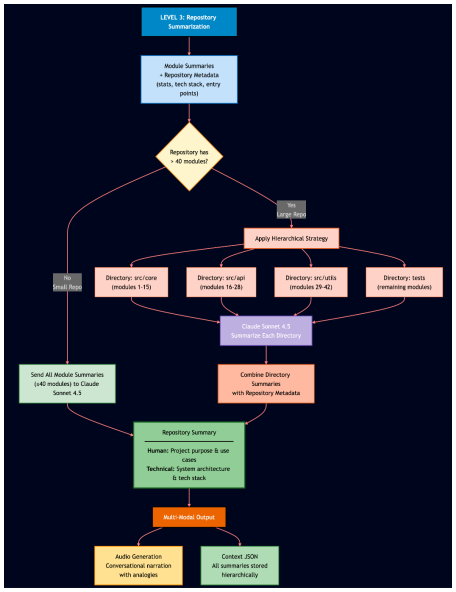


Figure 4: Repository summarization workflow

- **Architecture Visualization**: The Module Grouper uses Claude Sonnet's semantic understanding of module summaries to identify logical architectural groups beyond directory structure. It receives as input the module summaries (human and technical descriptions), module metadata from AST analysis (classes, imports, dependencies), and the repository structure. Instead of relying solely on directory structure, which may be flat, inconsistent, or organized by file type rather than functionality, the LLM analyzes module purposes and dependencies to create intuitive groupings based on common architectural categories like "Application Entry," "Business Logic," "Data Storage," "API Layer," "Authentication & Security," and "Utilities." For example, it might group "authentication.py, session.py, permissions.py" into "Security Layer" or "parser.py, lexer.py, tokenizer.py" into "Language Processing Core." This semantic grouping reveals the conceptual architecture that developers understand intuitively but isn't always reflected in file organization. If LLM grouping fails, the system falls back to directory-based grouping to ensure robustness.

  The Diagram Generator produces two types of visualizations from the logical groups and AST-extracted dependencies: ASCII dependency trees for terminal viewing and Mermaid.js flowcharts for documentation. The Mermaid diagrams use a semantic color-coding scheme where colors are automatically assigned based on group names, blue for entry points and main application logic, green for core components and APIs, purple for models and data storage, orange for network and adapters, red for authentication and security, yellow for utilities, gray for configuration and documentation, teal for extensions and plugins. This color scheme helps readers quickly identify functional areas without reading labels. The generator extracts dependencies by mapping import statements to actual files in the repository, handling both simple imports (import model) and dotted imports (from task_manager.task_handler import X). To prevent diagram overcrowding, the generator uses a simplified group-level view for all repositories, showing connections between logical groups rather than individual files, making architectural layers visible regardless of codebase size. Dependency arrows reveal architectural layers, such as UI components depending on Business Logic depending on Data Access, making layered architecture patterns visible. The generated .mmd files render in GitHub, VS Code, and documentation sites without requiring specialized tools, making architecture documentation accessible and maintainable. Figure 5 illustrates how modules are grouped semantically to produce the architecture diagram.

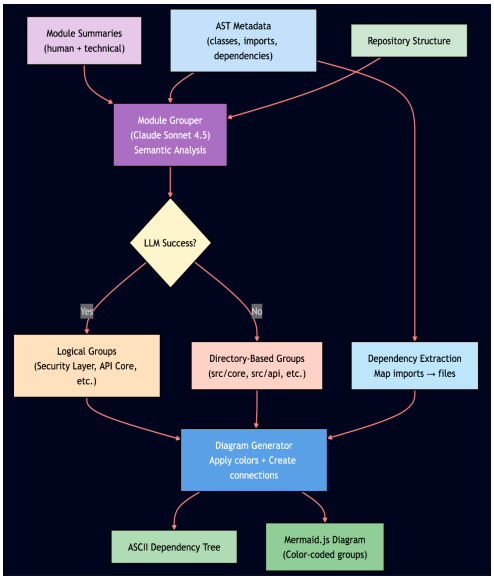


Figure 5: Architecture grouping and diagram generation

- **Audio Documentation Generation**: The Audio Generator converts repository summaries into conversational audio narrations, providing an accessible learning modality for both technical and non-technical audiences who need to understand what a codebase does without reading code or technical documentation. The generator uses the repository-level summary rather than individual module summaries because audio narrations work best as high-level introductions that provide conceptual understanding without overwhelming listeners with implementation details. Unlike reading documentation verbatim, the generator transforms the human-readable repository summary into an audio-optimized script that prioritizes comprehension and engagement through a two-stage process: first using Claude Sonnet 4.5 to rewrite the summary with audio-specific optimizations, then using Google Text-to-Speech (gTTS) to convert the script into MP3 format. The choice of gTTS provides free, unlimited text-to-speech conversion without API rate limits or costs, making the tool accessible to all users.

  The LLM transformation rewrites summaries for spoken delivery using several techniques. It uses conversational language that non-technical listeners can understand and adds real-world analogies to explain complex ideas like describing authentication as "a security checkpoint." Clear transitions help listeners follow along ("First..., Next..."), while acronyms are always spelled out since you can't pause audio to look things up. The script includes natural pauses (indicated by ellipses) where listeners need a moment to digest information, and opens with engaging hooks like "Imagine..." to grab attention immediately. To respect attention spans, narrations stay under two minutes (250-300 words), and content is framed as stories about solving problems rather than dry feature lists.

  After generating the script, gTTS converts it to MP3 with natural English pronunciation. The system saves both formats, audio for hands-free consumption, text script for accessibility tools and search. This multi-modal approach serves diverse audiences: managers understanding projects without technical expertise, product owners grasping system capabilities, newcomers building intuition before reading code, visually impaired individuals accessing documentation, and non-native speakers who find listening easier.

- **Dead Code Analyzer**: The Dead Code Analyzer combines AST-based structural detection with LLM-powered semantic validation to distinguish genuinely unused code from false positives. It receives dead code candidates from the AST Parser across six categories (unreferenced functions, unused classes, unused imports, unused global variables, unreachable code blocks, and suspicious patterns) and performs contextual analysis using Claude Sonnet 4.5 to determine their actual status.

  The analyzer uses a two-stage approach where the AST Parser identifies structural candidates based on reference counting and control flow analysis, but relies on the LLM to validate whether each candidate is

truly dead code. This separation matters because structural analysis cannot detect patterns like reflection-based method calls, framework callbacks, plugin registration, or public API methods intended for external library users. Also, not all candidates require LLM validation. The AST Parser assigns a needs_llm flag to each candidate, filtering out straightforward cases like unused imports that can be confidently identified through static analysis alone. Only candidates flagged as needs_llm: true proceed to semantic validation, which optimizes token usage by avoiding unnecessary API calls for obvious dead code patterns.

For each candidate requiring validation, the analyzer gathers rich context including the code snippet, docstring, class context (for methods), module context (whether it's in __init__.py or config files), and project context (library vs. application classification). This context enables the LLM to reason about usage patterns that pure static analysis would miss.

The system automatically detects whether the project is a library or application by analyzing patterns like __init__.py exports, __all__ declarations, CLI argument parsing imports, and entry point functions like main(). This distinction is critical because library projects expose methods as public API for external users even if they're never called internally, while application projects expect all code to be referenced somewhere within the codebase. The analyzer applies different validation rules accordingly, being conservative with libraries (marking exported class methods as false positives) but aggressive with applications (flagging unreferenced methods as dead code).

To optimize token usage, the analyzer batches candidates in groups of five per API call. Each validation returns a three-way classification (dead code, false positive, or uncertain) along with confidence scores, detailed reasoning, and actionable recommendations. The "uncertain" category allows edge cases to be flagged for human review rather than forcing binary decisions on ambiguous code patterns. The dead-code detection and validation workflow is shown in Figure 6 below.

Figure 6: Dead Code analysis workflow

Implementation

- **System Implementation**
    - **Technology Stack**: The implementation leverages:
        - Python AST Module: Built-in ast library for parsing Python source code into abstract syntax trees
        - Anthropic API: Claude Sonnet 4.5 (claude-sonnet-4-5-20250929) for all LLM-powered analysis
        - gTTS (Google Text-to-Speech): Converts text summaries to MP3 audio files
        - Mermaid.js: Generates color-coded architectural diagrams from structured data
        - Streamlit: Provides web-based user interface for repository analysis and result visualization
        - JSON: Stores intermediate results for reproducibility and resumable analysis
    - **Project Structure**:

```
repospeak/
├── src/
│   ├── ast_parser.py            # AST analysis and dead code detection
│   ├── function_summarizer.py   # Function-level summarization
│   ├── module_summarizer.py     # Module-level aggregation
│   ├── repo_summarizer.py       # Repository-level overview
│   ├── dead_code_analyzer.py    # LLM-powered validation
│   ├── module_grouper.py        # Semantic grouping
│   ├── diagram_generator.py     # Visualization generation
│   ├── audio_generator.py       # Speech synthesis
│   └── context_manager.py       # State management
├── config.py                    # API keys and configuration
├── repospeak.py                 # Main pipeline orchestrator
├── streamlit_app.py             # Web UI dashboard
├── evaluate_with_opus.py        # LLM-as-a-judge evaluation framework
├── audio/                       # Generated MP3 files and scripts
├── diagrams/                    # ASCII and Mermaid visualization files
├── ast_analysis_<repo>.json     # AST metadata per repository
├── context_<repo>.json          # Complete analysis state per repository
├── dead_code_analysis_<repo>.json # Validation results per repository
├── evaluation_<repo>.json       # Opus quality assessment per repository
└── summaries_<repo>.json        # All summaries per repository
```

    - **Configuration Management**: The config.py module loads environment variables from .env and provides system-wide settings:
        - ANTHROPIC_API_KEY: Claude API authentication
        - LLM_MODEL: Model identifier (defaults to claude-sonnet-4-5-20250929)
        - SKIP_PATTERNS: Directories to exclude during analysis (['test', 'tests', 'venv', '__pycache__', '.git', 'node_modules', '.pytest_cache'])
    - **Pipeline Orchestration (repospeak.py)**: The repospeak.py main script coordinates the complete analysis pipeline by instantiating components in sequence. It first runs the AST Parser to extract code structure and dead code candidates, then passes AST results to the Function Summarizer which processes each module's functions. The Module Summarizer receives function summaries and AST metadata to generate module-level descriptions, which feed into the Repository Summarizer for the final overview. In parallel with summarization, the Dead Code Analyzer validates AST candidates using LLM reasoning. After summarization completes, the Module Grouper creates logical architectural groups, the Diagram Generator produces visualizations, and

the Audio Generator converts the repository summary to speech. The Context Manager maintains state throughout, enabling each component to access previous results and persist outputs for the next stage.

- **Context Persistence (context_manager.py)**: The ContextManager class provides methods to update the hierarchical structure: store_ast_results() stores complete AST analysis, store_function_summaries() adds function summaries for a module, store_module_summary() adds module-level summaries, store_repo_summary() stores the repository overview, and store_dead_code_analysis() saves validation results. After each major pipeline stage, the in-memory context dict is persisted to disk via save_to_file() which writes the JSON to context_<repo >.json, it has this hierarchical structure as seen below and Figure 7 shows the overall context structure maintained during analysis.

```
{
  "repo_name": "repository-name",
  "ast_results": { /* per-file AST metadata */ },
  "function_summaries": { /* per-module function summaries */ },
  "module_summaries": { /* per-module aggregated summaries */ },
  "repo_summary": { "human": "...", "technical": "..." },
  "dead_code_analysis": { /* validation results */ },
  "logical_groups": { /* semantic architectural groups */ },
  "audio_narration": { /* optimized narration script */ }
}
```

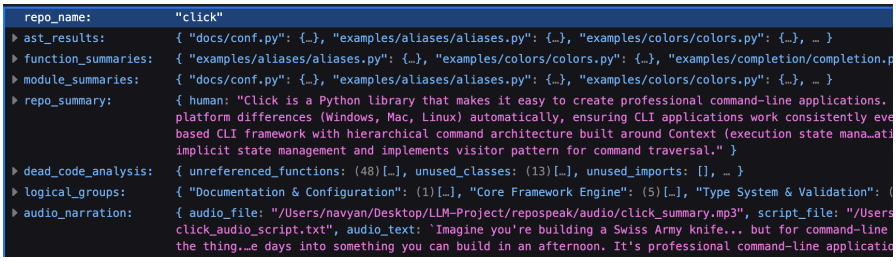


Figure 7: Repospeak context structure

This enables resumable analysis, if the pipeline crashes mid-execution, it can reload context and continue from the last completed stage rather than re-running expensive LLM calls.

- **AST Parser Implementation (ast_parser.py)**

  - **File Discovery and Parsing**: The CodeAnalyzer class uses Path.rglob("*.py") to recursively discover Python files, filtering out test directories and virtual environments using SKIP_PATTERNS. Each file is parsed with ast.parse() which converts source code to an Abstract Syntax Tree without execution. Parse errors are caught and logged but don't halt analysis.

  - **Metadata Extraction**: The parser uses ast.walk(tree) to traverse AST nodes and extract:
    - Functions: Captures signatures, type hints, decorators, and calls using FunctionDef and AsyncFunctionDef nodes. Full source code is extracted via ast.get_source_segment().
    - Classes: Extracts names, base classes from node.bases, and methods from nested FunctionDef nodes
    - Imports: Classifies import vs. from...import statements, distinguishing standard library from third-party
    - Global Variables: Identifies module-level Assign nodes to track variables like DEBUG = True or API_KEY = "..."

- **Dependency Graph**: After parsing all files, _build_dependency_graph() aggregates references into repository-wide sets: all_defined_functions, all_called_functions, all_defined_classes, all_instantiated_classes. This enables cross-module dead code detection.

- **Export Detection**: _detect_exports() identifies public API by parsing __init__.py from .module import Symbol patterns and finding __all__ declarations. Exported symbols are stored in self.exported_symbols and excluded from dead code candidates.

- **Dead Code Detection**: Six detection strategies are run by comparing defined vs. used symbols:
  - Unreferenced Functions: Finds functions that are defined but never called anywhere in the codebase. Excludes special methods, decorated framework functions, interface protocols, and exports
  - Unused Classes: Finds classes that are defined but never instantiated. Excludes exports
  - Unused Imports: Tracks symbol usage in calls, type hints, and assignments and flags zero-reference imports with needs_llm: false
  - Unused Variables: Detects write-only global variables
  - Unreachable Code: Identifies statements blocks after keywords like return/raise, if False:
  - Suspicious Patterns: Flags >6 parameters, >100 lines, single-call functions

Each candidate is tagged with needs_llm: true/false to optimize LLM validation.
The extracted AST metadata and detected dead-code candidates are shown in Figure 8 below.

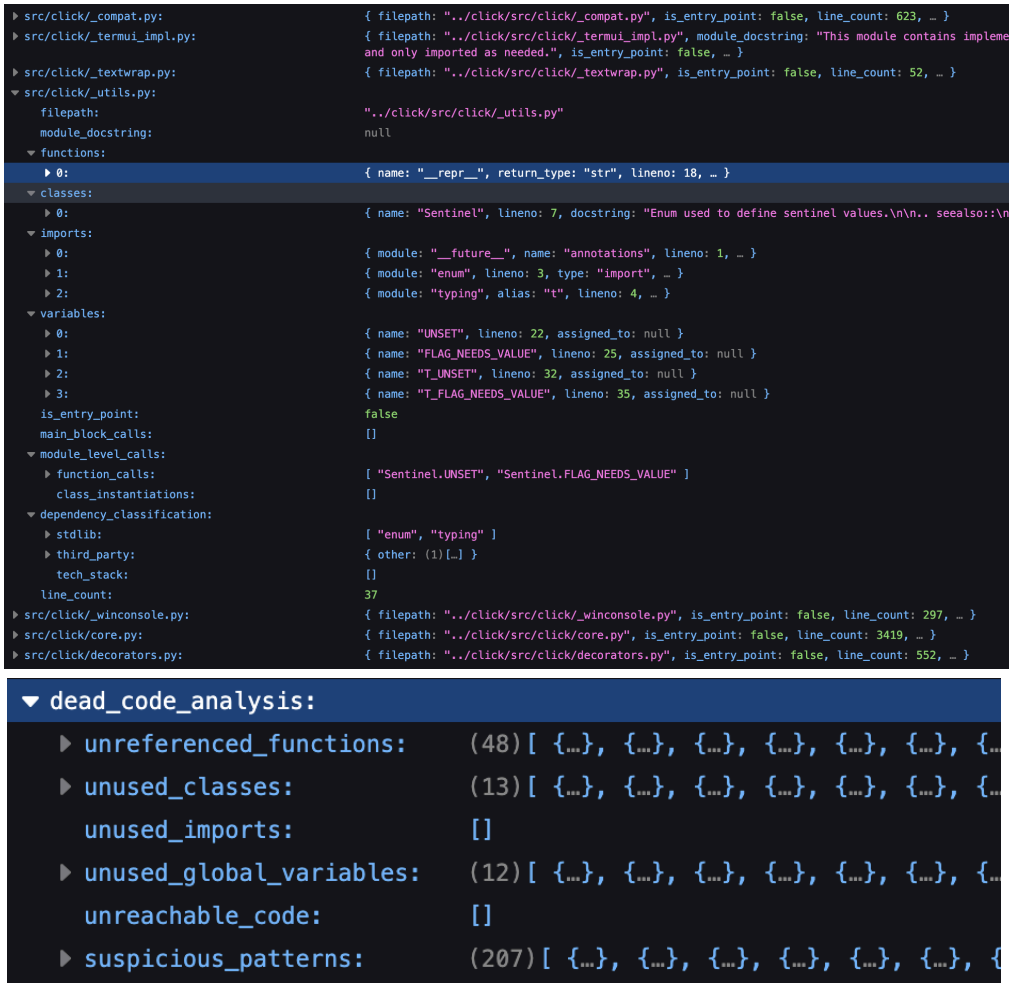


Figure 8: AST output and dead code candidates

- **Hierarchical Summarization Implementation**

    - **Function Summarizer Implementation (function_summarizer.py)**: The FunctionSummarizer class implements size-based routing using character count thresholds. The summarize_function() method checks len(func.get('code', '')) against MAX_FUNCTION_CHARS = 12000 and then decides whether full function summarization(_summarize_full_function) should be invoked or chunking (_summarize_with_chunking) should be invoked.

      For full function summarization, _summarize_full_function() builds a prompt string with formatted function metadata and makes a single Claude API call. API calls use temperature=0.2 for deterministic output and max_tokens=400 to enforce concise 5-6 line summaries.

      For chunked functions, _summarize_with_chunking() splits code by lines using code.split('\n'), creating overlapping segments with CHUNK_LINES = 60 and CHUNK_OVERLAP = 15. Each chunk is sent with extracted context (signature, key calls, variables) prepended. After receiving chunk summaries, a final aggregation call combines them using a prompt that lists all partial summaries and requests coherent integration.

      Response parsing uses _parse_two_level_summary() which scans for "Human Summary:" and "Technical Summary:" headers via string matching, accumulating subsequent lines into respective sections and stores the {"human": "...", "technical": "..."} dict into context manager. Figure 9 shows how function summaries are stored in the context.

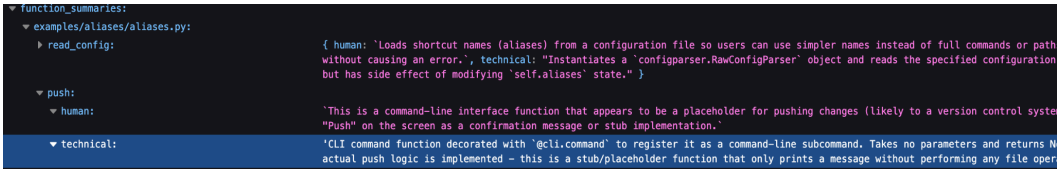


Figure 9: Stored function Summaries

    - **Module Summarizer (module_summarizer.py)**: The ModuleSummarizer class receives function summaries as a nested dict {module_path: {func_name: {"human": "...", "technical": "..."}}} along with AST metadata. The summarize_module() method implements size-based routing by checking len(function_summaries) against MAX_FUNCTIONS_PER_CALL = 30 and then decides whether full module summarization (_build_module_prompt()) should be invoked or large module summarization (_summarize_large_module()) should be invoked.

      For modules under the threshold, _build_module_prompt() constructs a prompt containing Module path, docstring, Class definitions formatted as class ClassName(BaseClass) with method lists, Import dependencies extracted from AST (standard library vs. third-party) along with function summaries and makes a single Claude API call.

      For large modules, _summarize_large_module() uses chunks = [list(items)[i:i+30] for i in range(0, len(items), 30)] to batch functions into groups of 30. Each group is summarized via separate API call, producing intermediate summaries. A final aggregation call combines group summaries with the prompt: "Combine these group summaries into coherent module-level summary..."

      For modules without functions (e.g., config files, __init__.py), _summarize_module_from_code() fallback extracts the module docstring and variable definitions directly from AST, generating minimal summaries from available metadata.

      All API calls use temperature=0.2 and max_tokens=500 (vs. 400 for functions) to accommodate longer module-level descriptions. Response parsing reuses the same _parse_two_level_summary() method and stores the module summaries into the context manager. Figure below shows the module summaries dict stored in context manager and chunking for bigger modules. The module

summarization results and chunking for large modules are illustrated in Figure 10 below.

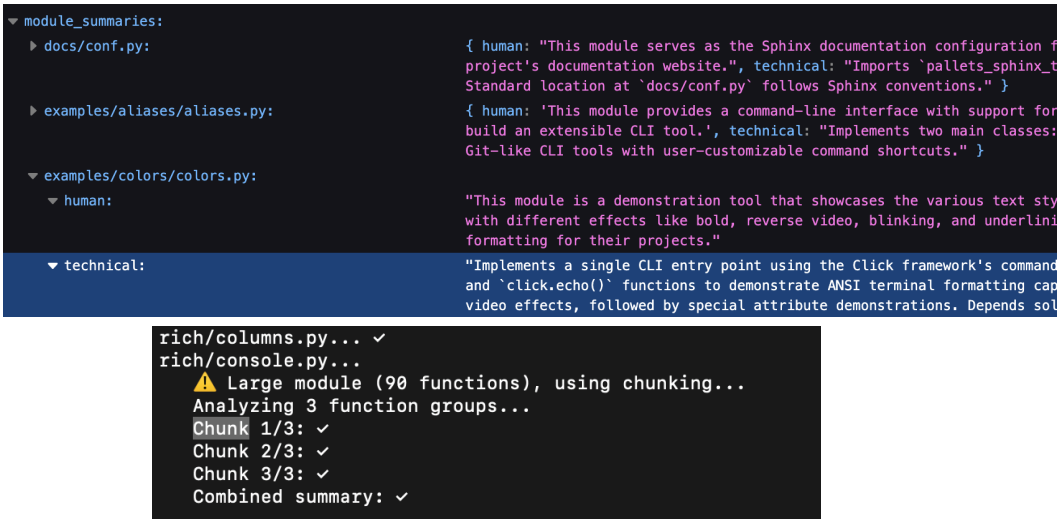


Figure 10: Stored module summaries and Chunking of large modules

- **Repository Summarizer (repo_summarizer.py)**: The RepoSummarizer class receives module summaries as {module_path: {"human": "...", "technical": "..."}} and AST statistics dict containing totals. The summarize_repository() method implements size-based routing by checking len(module_summaries) against MAX_MODULES_PER_CALL = 40 and then decides whether full module summarization (_build_repo_context()) should be invoked or large module summarization (_summarize_large_repo) should be invoked.

  For repositories under the threshold, _build_repo_context() constructs repository metadata using AST results like total modules, total functions, total classes, entry points using a single Claude API call.

  For large repositories, _summarize_large_repo() groups modules by directory using pathlib.Path(module_path).parent. Each directory group is summarized via a separate API call, producing directory-level summaries. A final aggregation call combines directory summaries with repository metadata.

  Project type detection uses heuristics on AST data: checks for if __name__ == "__main__" blocks, CLI parsing imports (argparse, click), and __init__.py export patterns to classify as library vs. application. This classification is passed to the prompt via context variable but doesn't change the fundamental summarization logic, instead changing only the LLM's interpretation.

  API calls use temperature=0.2 and max_tokens=600 (higher than module/function limits) to accommodate comprehensive repository descriptions. Response parsing uses _parse_two_level_summary() method and stores the repository summary into the context manager. Figure 11 presents the repository-level summary produced by RepoSpeak.

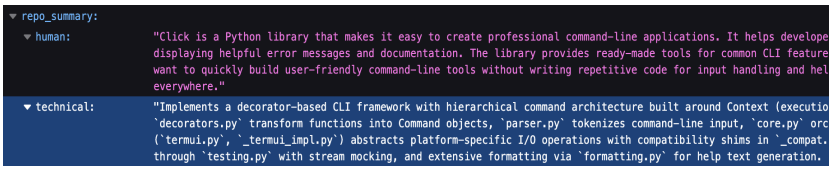


Figure 11: Repository summary output

- **Architecture Visualization Implementation**

    - **Module Grouper (module_grouper.py)**: The ModuleGrouper class receives module summaries as a dict mapping paths to dual summaries. The group_modules() method constructs a prompt listing each module path with its human summary, then requests JSON-formatted grouping from Claude. The prompt includes suggested architectural categories (Application Entry, Business Logic, Data Storage, API Layer, etc.) and specifies output format as a JSON object with category names as keys and module path arrays as values.

      The implementation makes a single API call with max_tokens=2000 to accommodate grouping responses for repositories with dozens of modules. Response parsing handles markdown code blocks by detecting triple-backtick delimiters and extracting JSON content between them. The parser validates that all input modules appear in exactly one group by comparing sets of input modules versus grouped modules, adding any missing modules to an "Other" category.

      If the API call fails or JSON parsing fails, the _fallback_grouping() method extracts the first directory component from each module path using module_path.split('/')[0] and groups modules by these directories. Root-level files without directory prefixes are grouped under "Root Modules." This ensures the system produces visualizations even when LLM grouping is unavailable.

    - **Diagram Generator (diagram_generator.py)**: The DiagramGenerator class receives logical groups and AST results, then generates two diagram types: ASCII dependency trees for terminal viewing and Mermaid.js flowcharts for documentation.

      Dependency extraction builds a lookup table mapping module stems, full paths without extensions, and dotted-path versions to actual file paths. For each module's imports from AST, the implementation attempts to resolve import statements to repository files by checking the lookup table. Simple imports like import model resolve via stem matching, while dotted imports like from task_manager.task_handler import X resolve by converting dots to slashes and checking path matches. This produces a dependency dict mapping each file to the set of files it imports.

      For ASCII output, _generate_ascii_dependencies() iterates through modules alphabetically, printing each module name followed by indented dependency lists using Unicode box-drawing characters for visual tree structure.

      For Mermaid output, the implementation always uses a simplified group-level view regardless of repository size. The _generate_simplified_mermaid() method aggregates file-level dependencies into group-level connections by checking which groups contain dependent files. The color assignment function get_group_color() uses keyword matching on lowercase group names by checking for terms like "entry," "main," "core," "api," "model," "data," "auth," "security," "util" and returns hex color codes. Blue indicates entry points, green for core/API, purple for models, orange for network components, red for security, yellow for utilities, gray for configuration, teal for plugins. The default color is neutral gray for unmatched groups.

      The Mermaid syntax generation creates a graph TD (top-down flowchart) with subgraphs for each logical group. Nodes represent groups with assigned colors applied via style directives. Dependency arrows connect groups using Group1 --> Group2 syntax and the output is saved to .mmd files in the diagrams/ directory via the _save_diagram() method. The logical module groups are shown in Figure 12 below.

Figure 12: Logical groups generated by Sonnet

- **Audio Documentation Implementation (audio_generator.py)**
    - **Script Optimization with LLM**: The AudioGenerator class receives the human-readable repository summary and implements a two-stage transformation process. The create_audio_optimized_summary() method constructs a prompt containing the repository name, original summary, and eight audio-specific requirements: conversational tone, analogies, clear transitions, acronym explanations, natural pauses, engaging hooks, length constraints (250-300 words), and storytelling framing.

      The prompt includes a before-after example demonstrating the transformation style. For instance, a dry description like "Flask web application with user authentication" becomes an analogy-rich narration about "a digital doorway with security checkpoints." This example guides Claude's output style without requiring explicit instruction on every transformation technique.

      The implementation makes a single API call with temperature=0.7 (higher than the 0.2 used for code summarization) to encourage creative analogies and conversational phrasing while maintaining accuracy. The max_tokens=1000 limit enforces the length constraint, keeping narrations under the target 250-300 word range for approximately two-minute audio duration.

    - **Text-to-Speech Conversion**: The text_to_speech() method uses gTTS to convert the optimized script to MP3 format. The implementation instantiates a gTTS object with parameters text, lang='en', and slow=False for natural speech pace, then saves the audio to the audio/ directory with the repository name as filename.

      gTTS operates entirely offline once initialized, making unlimited API calls without rate limits or costs. The library handles pronunciation, inflection, and pacing automatically based on English language models.

    - **Dual Output**: The generate_repository_audio() orchestrator method saves both formats: the audio-optimized text script is written to a .txt file for accessibility tools and searchability, while the MP3 is saved for hands-free consumption. The implementation calculates word count and estimates duration using a 150 words-per-minute speaking rate, returning these statistics along with file paths in a dict structure for storage in the context manager.

      The method saves the audio-optimized script to audio/<repo>_audio_script.txt and the MP3 to audio/<repo>_summary.mp3. Both files are stored in the audio/ directory, with the text script enabling screen reader access, search indexing, and providing a reference transcript for users who want to review the exact wording before listening. Figure 13 illustrates the audio-narration transcript that is generated.

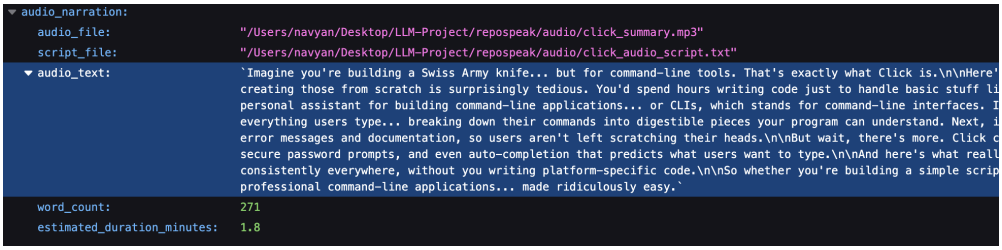


Figure 13: Audio narration transcript stored in context

- **Dead Code Analyzer Implementation (dead_code_analyzer.py)**

    - **Candidate Filtering**: The DeadCodeAnalyzer class receives the complete AST results dict containing a dead_code_candidates section from the analysis summary. The analyze_dead_code() method iterates through six candidate categories (unreferenced functions, unused classes, unused global variables, unreachable code, suspicious patterns) and filters each by the needs_llm flag using list comprehension: needs_llm = [c for c in candidates if c.get('needs_llm', True)]. Candidates with needs_llm: false (primarily unused imports) are skipped entirely, avoiding unnecessary API costs for definitively unused code.

    - **Context Gathering**: For each candidate requiring validation, _gather_context() constructs a rich context dict by extracting information from multiple AST levels. Project context comes from the __analysis_summary section (total modules, functions, classes) plus library detection. Module context checks if the file path ends with __init__.py or config file patterns (conf.py, setup.py, conftest.py).

      For function candidates, _get_function_context() searches the module's functions list for matching names, extracts code and docstring, then checks if the function appears in any class's methods list to determine if it's a class method. If so, it gathers class context including base classes, all methods, and whether the class is exported.

    - **Library vs. Application Detection**: The _is_library_project() method implements a scoring system that accumulates evidence for both project types. It iterates through all modules counting application indicators (main functions in root files worth 5 points, CLI parsing imports worth 3 points) and library indicators (__init__.py with relative imports worth 2 points, __all__ declarations worth 3 points). Special rules apply: multiple exporting __init__.py files add 5 bonus library points, while a single main function in utility files reduces application score. The final classification compares scores: is_library = library_indicators > application_indicators.

    - **Batch Processing**: The analyzer processes candidates in batches of 5 using range(0, len(needs_llm), 5) to create batch slices. For each batch, _analyze_batch() constructs a prompt listing all items with their contexts, sends a single API call, and expects a JSON array response with one analysis object per item. API calls use temperature=0.2 for consistent reasoning and max_tokens=4000 to accommodate detailed explanations for multiple candidates in each batch.

      The prompt includes conditional logic based on project type, instructing Claude to be conservative with libraries (flag exported methods as false positives) but aggressive with applications (flag unreferenced methods as dead code). It specifies three valid status values (dead_code, false_positive, uncertain) and requests structured output with category, confidence score (0-100), reasoning, and recommendation (keep, safe_to_delete, investigate, mark_deprecated).

    - **Response Parsing and Validation**: The _parse_llm_response() method handles markdown code blocks by splitting on triple-backtick delimiters, then parses JSON. The implementation merges LLM analysis with original candidate data, creating result dicts that contain both AST information (module path, line number) and LLM validation (status, confidence, reasoning). If JSON parsing fails or the API calls errors, the method returns fallback dicts with status: 'uncertain' and error

messages to ensure the pipeline continues despite validation failures.

The results of the LLM-based dead-code validation are presented in Figure 14 below.

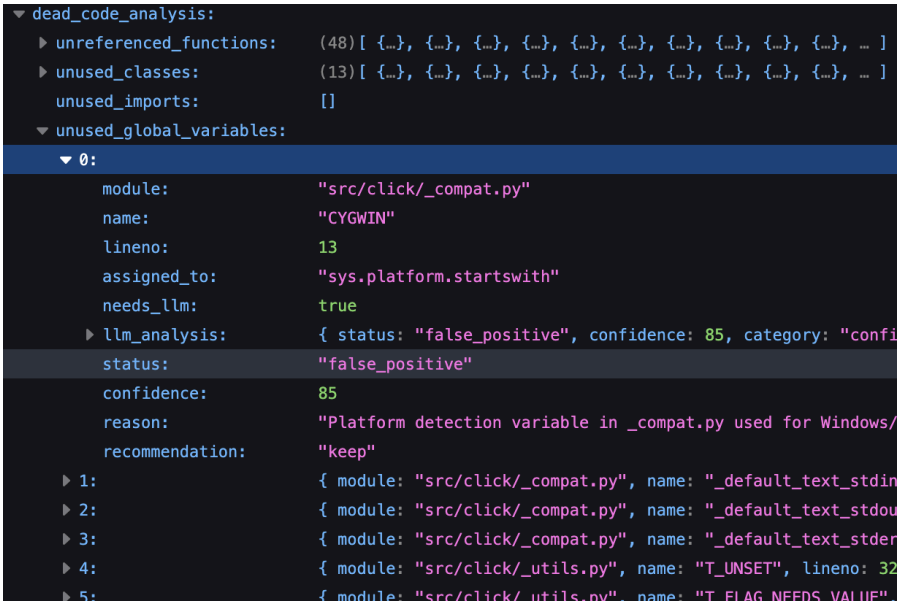


Figure 14: Dead code validation results stored in context

- **Web UI (streamlit_app.py)**: The Streamlit application provides an interactive web dashboard for visualizing analysis results. The application uses st.set_page_config() for layout configuration and injects custom CSS via st.markdown() for styled stat boxes and color-coded severity indicators.

  Context loading uses Path(".").glob("context_*.json") to discover all available analysis results, extracting repository names from filenames for the sidebar selector. When a repository is selected, the application loads its context JSON and displays six main sections:

  - Repository Overview (display_overview()): Displays metrics using Streamlit columns with st.columns(3) for files, functions, and classes counts. Repository summary is shown with collapsible technical details using st.expander().
  - Audio Narration (display_audio()): Checks for MP3 files in the audio/ directory, displays word count and duration metrics, embeds the audio player with st.audio(audio_bytes, format='audio/mp3'), and shows the transcript in a text area.
  - Architecture Diagram (display_architecture()): Reads .mmd files and generates Mermaid Live Editor links by base64-encoding the diagram code into URL parameters. The link enables interactive viewing with pan/zoom. Logical groups are displayed as expandable sections showing module lists.
  - Module Dependencies (display_dependencies()): Loads ASCII dependency trees from diagrams/ directory and displays them in code blocks with st.code(deps_content, language="text").
  - Dead Code Analysis (display_dead_code()): Iterates through all six dead code categories, displays summary metrics in a six-column layout, and creates expandable sections for each category. For each candidate, it extracts status, confidence, recommendation, and reasoning from the LLM analysis, applies color-coded status icons (red circle for dead code, green for false positive, yellow for uncertain), and formats the information with markdown.
  - Interactive Chat: Implements a conversational interface where users can ask questions about the codebase. The system uses Claude with the loaded context JSON as part of the system prompt, enabling accurate responses grounded in the actual repository analysis. Chat history is maintained in Streamlit session state (st.session_state['messages']) and displayed with alternating user/assistant message bubbles. In addition, the chat interface implements an LLM-based intent

parser (parse_intent()) that analyzes user queries and routes them to 14 specialized handlers (e.g., search_functions, search_dead_code, get_module_summary). Module and function summaries include real-time keyword search via st.text_input() with case-insensitive filtering that displays only matching results in expandable sections.

Experimental Setup

- **Test Repositories**

  We evaluated RepoSpeak on six Python repositories spanning different sizes, domains, and architectural patterns to demonstrate the system's ability to handle diverse codebases. The repositories were selected to represent a range from small educational projects to large production libraries:

Repository	Domain	Modules	Functions	Classes	Description
test_project	Educational	6	35	4	Small sample project for initial testing
gnn_training_master	Machine Learning	5	4	2	Graph Neural Network training framework
requests	HTTP client	20	185	45	Popular Python HTTP library
click	CLI Framework	31	347	78	Command-line interface creation toolkit
flask	Web Framework	34	281	52	Lightweight WSGI web application framework
rich	Terminal UI	123	574	195	Rich text and beautiful formatting in terminal

  The repositories range from 5 modules (gnn_training-master) to 123 modules (rich), and from 4 functions (gnn_training-master) to 574 functions (rich), providing comprehensive coverage of small, medium, and large codebases. Most repositories are open-source projects with well-documented public APIs, enabling validation of the system's ability to distinguish public interfaces from dead code.

- **Evaluation Methodology**

  We evaluated RepoSpeak's performance through two complementary approaches:

  - **Performance Metrics Collection**: Each repository was processed through the complete RepoSpeak pipeline (AST parsing, hierarchical summarization, dead code detection, architecture visualization, and audio generation) on a MacBook running macOS Tahoe 26.0.1. The system used Claude Sonnet 4.5 (claude-sonnet-4-5-20250929) for all LLM operations. We measured quantitative metrics including analysis completion time, API call counts, and dead code detection statistics across all six repositories.
  - **LLM-Based Quality Evaluation**: We implemented an automated evaluation framework (evaluate_with_opus.py) that uses Claude Opus 4.5 as an independent judge to assess output quality. The framework sends evaluation prompts to Opus with structured JSON schemas for consistent rating formats, collecting scores on 1-5 scales for each quality dimension. For each repository, Opus evaluates function summaries, module summaries, repository summaries, architecture diagrams, dead code classifications, and audio transcripts across multiple quality dimensions. Results are saved to evaluation_<repo>.json files containing overall scores, dimension-specific ratings, identified errors, and improvement recommendations. This approach provides systematic quality assessment without requiring manual review.

- **Metrics**

  We measured system performance across three categories:

  - **Quantitative Metrics**:

    - Analysis completion time per repository
    - Dead code candidates detected (by category)
    - LLM Dead Code Validation Results (dead code vs. false positive vs. uncertain ratios)
    - API call counts per repository

  - **Quality Metrics (via Claude Opus evaluation):** For all quality evaluations, Opus received the same inputs originally provided to Sonnet alongside Sonnet's generated outputs, enabling independent assessment of accuracy and quality.

    - **Function/Module/Repository Summaries**:

      - Inputs: Original source code (or AST metadata for modules/repos) + Sonnet-generated summary
      - Evaluation: Rated on 1-5 scale for Factual Accuracy, Completeness, Clarity, and Technical Depth
      - Outputs: Overall scores, error lists, missing elements, and recommendations

    - **Architecture Diagram**:

      - Inputs: List of repository files + Mermaid diagram code + Logical groups dict generated by Sonnet
      - Evaluation: Completeness (all major modules included), Logical Grouping (semantic sense), Diagram Quality (hierarchy, clutter)
      - Outputs: Ratings per criterion and suggestions for improvements

    - **Audio Transcripts**:

      - Inputs: Original repository summary generated by Sonnet + Audio-optimized transcript + Word count
      - Evaluation: Accuracy, Analogies (count/quality), Accessibility, Engagement, Listenability
      - Outputs: Ratings per dimension and suggestions

    - **Dead Code Classifications**:

      - Inputs: Code snippet + Module context + Sonnet's classification (status, confidence, reasoning, evidence)
      - Evaluation: Opus makes independent classification, compares with Sonnet's decision
      - Outputs: Agreement rates (percentage of times Opus agrees with Sonnet's classification), calculated separately for each dead code category (unreferenced functions, unused classes, etc.) and for each status type (dead_code, false_positive, uncertain).

- **Configuration**

  All experiments used consistent configuration:

  - **Generation (RepoSpeak System)**:
    - LLM Model: Claude Sonnet 4.5 (claude-sonnet-4-5-20250929)
    - Temperature: 0.2 for summarization and dead code analysis, 0.7 for audio generation
    - Max Tokens: 400 (functions), 500 (modules), 600 (repositories), 1000 (audio), 4000 (dead code batches)
    - Chunking Thresholds: 200 lines (functions), 30 functions (modules), 40 modules (repositories)
    - Dead Code Batch Size: 5 candidates per API call
    - Skip Patterns: ['test', 'tests', 'venv', '__pycache__', '.git', 'node_modules', '.pytest_cache']
  - **Evaluation (Quality Assessment)**:
    - LLM Model: Claude Opus 4.5 (claude-opus-4-5-20251101)
    - Temperature: 0 (for deterministic, consistent scoring)
    - Max Tokens: 2000-3000 (depending on evaluation type)

  Using a different, more capable model (Opus) for evaluation ensures unbiased quality assessment of Sonnet-generated outputs, following the LLM-as-a-judge methodology where the evaluator is independent from the production system.

  Results were stored as JSON files (context_<repo>.json, ast_analysis_<repo>.json, dead_code_analysis_<repo>.json) and evaluation scores in evaluation_<repo>.json, enabling reproducibility and detailed analysis. Figure 15 shows the evaluation output generated by Claude Opus.

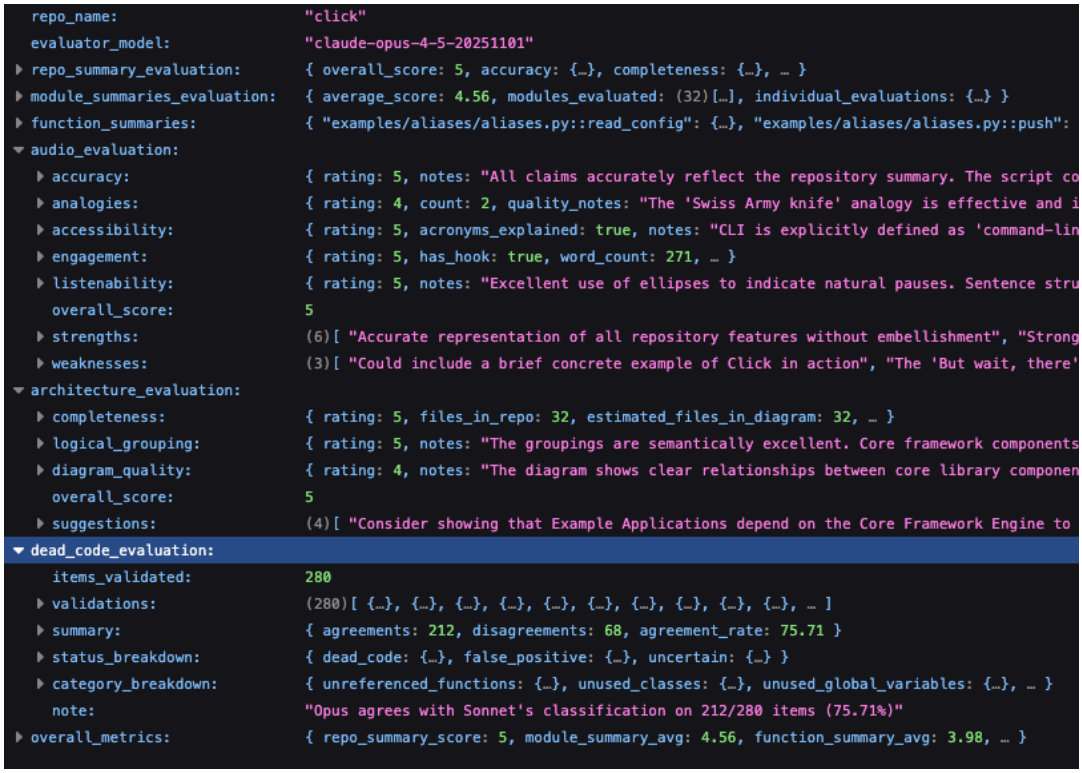


Figure 15: LLM-As-a-Judge Evaluation output by Opus

Results and Analysis

- **Quantitative Performance metrics**
    - **Analysis Completion Time**

Repository	Modules	Functions	Dead code candidates analyzed	Processing Stage
test_project	6	35	27	~ 6 -7 minutes
gnn_training_master	5	4	1	~ 1-2 minutes
requests	20	185	175	~ 30 - 40 minutes
click	31	347	280	~ 60 - 70 minutes
flask	34	281	318	~ 50 - 60 minutes
rich	123	574	1125	~ 150 -180 minutes

Processing time scales non-linearly with repository complexity. The gnn_training-master repository completed in 1-2 minutes due to minimal function count (4) and only a single dead code candidate requiring validation. In contrast, the rich library required 150-180 minutes despite having only 1.65x more functions than click (574 vs. 347), primarily due to its extensive module count (123) and dead code validation burden (1,125 candidates vs. 280 for click).

The flask and click repositories demonstrate that function count alone does not determine processing time. Flask, with 66 fewer functions than click (281 vs. 347), completed 10-20 minutes faster, yet both repositories have similar module counts (34 vs. 31). This suggests that dead code complexity, particularly the number of candidates flagged as "suspicious patterns", impacts processing time more significantly than raw function volume.

The requests library, despite having moderate size (20 modules, 185 functions), required 30-40 minutes due to 175 dead code candidates. The high proportion of false positives in requests (97.1%) indicates that many API methods and library integration points were initially flagged but required semantic validation to classify correctly, adding substantial processing overhead.

Overall, the system demonstrates acceptable scalability: small repositories (< 50 functions) complete in under 10 minutes, medium-sized projects (150-350 functions) finish within an hour, and large libraries (500+ functions) require 2-2.5 hours. The dominant performance bottleneck is dead code validation, which consumes approximately 60-70% of total processing time for repositories with extensive suspicious pattern detection (rich, click, flask).

    - **Dead Code Candidates by Category**

The AST parser identified 1,926 total dead code candidates across all six repositories, distributed across six detection categories: Suspicious Patterns, Unreferenced Functions, Unused Classes, Unused Global Variables, Unreachable Code, Unused Imports. The distribution varies significantly across repositories based on their maturity and purpose:

Repository	Unreferenced Functions	Unused Classes	Unused Globals	Unreachable Code	Suspicious Patterns	Total
test_project	15	1	4	3	4	27
gnn_training_master	0	0	1	0	0	1
requests	21	9	13	0	132	175
click	48	13	12	0	207	280
Flask	60	31	23	0	204	318
rich	66	53	37	0	969	1125
Total	210 (10.9%)	107 (5.6%)	90 (4.7%)	3 (0.2%)	1516 (78.7%)	1926

Suspicious patterns dominate (78.7%), primarily reflecting code complexity in mature libraries rather than true dead code. The rich library alone contributed 969 suspicious patterns, indicating many utility functions with extensive parameter lists or long implementations. This category serves more as a code quality indicator than a dead code detector.

Unreferenced functions and unused classes show higher concentrations in library projects (flask, rich, click) where many symbols represent public APIs or framework hooks not called internally.

- **LLM Dead Code Validation Results**

  All 1,926 candidates underwent semantic validation with Claude Sonnet 4.5, which classified each item into three categories: dead_code (genuinely unused), false_positive (legitimate code incorrectly flagged), or uncertain (requires human judgment):

Repository	Total	DeadCode	False Positive	Uncertain
test_project	27	22 (81.5%)	1 (3.7%)	4 (14.8%)
gnn_training_master	1	1 (100%)	0	0
requests	175	3 (1.7%)	170 (97.1%)	2 (1.1%)
click	280	30 (10.7%)	200 (71.4%)	50 (17.9%)
flask	318	3 (0.9%)	278 (87.4%)	37 (11.6%)
rich	1125	61 (5.4%)	851 (75.6%)	213 (18.9%)
Overall	1926	120 (6.2%)	1500 (77.9%)	306 (15.9%)

The 77.9% false positive rate demonstrates the critical value of semantic validation. Without LLM analysis, static analysis alone would have incorrectly flagged 1,500 legitimate code constructs as dead code. The false positives primarily consist of public API methods, framework integration points, and backward compatibility interfaces that appear unused within the repository but serve

external consumers.

The stark contrast between test_project (81.5% dead code) and mature libraries (1-11% dead code) reflects project purpose: test_project contains intentional dead code examples, while production libraries maintain clean codebases. The uncertain category (15.9%) represents edge cases like deprecation candidates and plugin extension points requiring human judgment.

- **API Call Efficiency**

  The system optimized API usage through strategic batching across different pipeline stages:

Repository	Function Calls	Module Calls	Repo Calls	Dead Code Calls	Total API Calls
test_project	35	6	1	6	48
gnn_training_master	4	5	1	1	11
requests	185	20	1	35	241
click	347	31	1	56	435
flask	281	34	1	64	380
rich	574	123	4	225	926
Total	1426	219	9	387	2041

The batching strategy significantly reduced API overhead: functions are processed individually (1 per call), modules batch up to 30 function summaries, repositories batch up to 40 module summaries, and dead code validation processes 5 candidates per call. This compressed 3,571 potential individual calls into 2,041 actual calls, a 43% reduction.

Function summarization accounts for 70% of total API calls, while dead code validation (19%) and module/repository summarization (11%) benefit from aggressive batching. The system scales linearly with repository size, requiring <50 calls for small projects and ~900 calls for large libraries like rich.

- **Summary Quality Evaluation**

Claude Opus 4.5 evaluated summaries across three hierarchical levels (function, module, repository) using four quality dimensions: Factual Accuracy, Completeness, Clarity, and Technical Depth. Each dimension was rated on a 1-5 scale, with an overall score calculated per summary.

Repository	Repository Summary	Module Summaries Average	Function Summaries Average
test_project	5.0	4.57	4.53
gnn_training_master	5.0	4.17	4.60
requests	5.0	4.43	4.42

click	5.0	4.56	3.98
flask	5.0	4.46	4.20
rich	4.0	4.42	4.38
Average	4.83	4.43	4.35

Repository-level summaries achieved the highest scores (4.83 average), with five repositories earning perfect 5.0 ratings. The rich library received a 4.0 due to minor completeness issues in capturing all architectural details across its 123 modules. Opus noted that while the summary accurately described the terminal rendering architecture, it omitted specific details about certain subsystems like the console protocol and style management.

Module summaries averaged 4.43, demonstrating consistent quality across all repositories. The gnn_training-master repository scored lowest (4.17) due to limited module complexity, with only 5 modules and minimal functionality, there was less semantic content to summarize, leading to brevity concerns. Conversely, test_project and click scored highest (4.57, 4.56), benefiting from well-structured modules with clear separation of concerns.

Function summaries averaged 4.35, showing slightly more variation. Click scored lowest (3.98) despite having high module scores, primarily due to its extensive use of decorators and complex parameter handling that challenged accurate summarization. Several function summaries incorrectly described decorator effects or omitted optional parameter details. The gnn_training-master repository scored highest (4.60) due to its simple, straightforward functions with minimal complexity.

Summaries excelled in clarity (4.8/5 average) and factual accuracy (4.6/5 average). Opus consistently praised the dual-format approach, separating human-readable explanations from technical details for effectively serving both developers and non-technical stakeholders. The main weakness was completeness (4.2/5 average), where summaries sometimes missed edge cases, unused parameters, or deprecated features that developers might need to know about.

- **Architecture Correctness Evaluation**

Opus evaluated architecture diagrams across three dimensions: Completeness (all major modules included), Logical Grouping (semantic coherence of groups), and Diagram Quality (visual hierarchy and clarity).

Repository	Completeness	Logical Grouping	Diagram Quality	Overall Score
test_project	5	5	4	5
gnn_training_master	5	4	3	4
requests	5	5	4	5
click	5	5	4	5
flask	5	5	3	4
rich	5	5	4	5
Average	5	4.83	3.67	4.67

All six repositories achieved perfect completeness scores (5.0), indicating that the Module Grouper and Diagram Generator successfully identified and included all major modules. Opus verified that no significant files were omitted from the visualizations, validating the AST parser's file discovery and filtering logic.

Logical grouping scored 4.83 on average, with five repositories receiving perfect 5.0 ratings. The gnn_training-master repository scored slightly lower (4.0) because its small size (5 modules) resulted in overly granular grouping, each module becoming its own group rather than forming meaningful semantic clusters. Opus noted this was acceptable given the limited module count but suggested a minimum threshold for grouping.

Diagram quality averaged 3.67, the lowest of the three dimensions. Flask and gnn_training-master scored 3.0 due to visual clarity issues: Opus identified that package initialization files (__init__.py) showed speculative dependencies to other modules that don't actually exist in practice, adding visual clutter. The remaining repositories scored 4.0, with Opus praising clean hierarchies and color-coded semantic groups but suggesting improvements like nested subgraphs to show package structure more clearly.

The overall architecture scores (4.67 average) demonstrate that the semantic grouping approach successfully creates meaningful visualizations. The LLM-based Module Grouper correctly identified logical boundaries (e.g., "Business Logic," "Data Storage," "Utilities") rather than defaulting to directory-based grouping, which Opus noted would have been less intuitive for understanding system architecture.

- **Audio Narration Quality Evaluation**

Opus evaluated audio transcripts across five dimensions: Accuracy (factual correctness), Analogies (count and quality), Accessibility (jargon-free language), Engagement (hooks and word count), and Listenability (natural speech patterns). Overall scores represent holistic quality judgment rather than arithmetic averages of individual dimensions.

Repository	Accuracy	Analogies	Accessibility	Engagement	Listenability	Overall
test_project	5	5	4	5	5	5
gnn_training_master	5	5	5	5	5	5
requests	5	5	5	5	5	5
click	5	4	5	5	5	5
flask	5	5	4	5	5	5
rich	5	5	5	5	5	5
Average	5	4.83	4.67	5	5	5

All repositories achieved perfect overall scores (5.0), demonstrating exceptional quality for conversational narration despite minor weaknesses in individual dimensions. The higher temperature setting (0.7) for audio generation successfully produced engaging, natural-sounding content.

Accuracy scored perfectly (5.0), with Opus verifying that all transcripts represented repository features without hallucinations. Analogies averaged 4.83, with 2-6 analogies per transcript. Click scored 4.0 with only 2 analogies, while flask excelled with 6 high-quality metaphors.

Accessibility averaged 4.67, with test_project and flask receiving 4.0 for using technical terms ("terminal," "hard drive") without explanation. The remaining repositories achieved 5.0 by explaining or avoiding jargon entirely.

Engagement and listenability both achieved perfect 5.0 scores, with all transcripts hitting the target length (250-300 words) and employing compelling hooks and natural speech patterns.

- **Dead Code Detection Evaluation**

To validate Sonnet's dead code classifications, Opus independently evaluated the same candidates and compared decisions. Agreement rates measure how often Opus concurred with Sonnet's classification (dead_code, false_positive, or uncertain).

Repository	Items Validated	Agreements	Disagreements	Agreement Rate
test_project	27	18	9	66.7%
gnn_training_master	1	1	0	100%
requests	175	167	8	95.4%
click	280	212	68	75.7%
flask	318	288	30	90.6%
rich	1125	852	273	75.7%
Total	1926	1538	388	79.9%

The overall agreement rate of 79.9% demonstrates substantial consistency between two independent LLM evaluators on semantic dead code detection. Agreement was highest for production libraries (requests: 95.4%, flask: 90.6%) where false positives dominate, both models correctly identified most flagged code as legitimate library APIs. Test_project showed lower agreement (66.7%) due to its mix of genuine dead code and edge cases designed for teaching purposes.

- **Overall Results**

RepoSpeak achieved an overall quality score of 4.57/5.0 across all six repositories, with audio narration scoring highest (5.0) and function summaries showing the most variation (4.37). Dead code agreement between Sonnet and Opus averaged 84.0%, demonstrating substantial inter-model consistency.

Individual repository scores ranged from 4.43 (rich) to 4.74 (requests). Requests excelled due to clean architecture and high dead code agreement (95.4%), while rich's massive scale (123 modules, 1,125 dead code candidates) presented greater challenges. Dead code agreement varied from perfect (gnn_training-master: 100%) to moderate (test_project: 66.7%), reflecting differences in code complexity and ambiguity.

The consistency across diverse repositories, from 5-module examples to 123-module production libraries,

demonstrates RepoSpeak's robustness and scalability. All repositories exceeded 4.4 overall quality, validating the hybrid AST + LLM approach for codebases of varying sizes and complexity levels.

Discussion

- **Strengths**: RepoSpeak demonstrates several key strengths that validate the hybrid AST + LLM approach for repository documentation:
    - **Effective false positive reduction**: Semantic validation correctly identified 77.9% of AST-flagged candidates as legitimate code (library APIs, framework hooks), preventing the massive false alarm rate that pure static analysis would produce. This demonstrates that LLM semantic understanding significantly improves precision over syntactic approaches alone.
    - **Scalable hierarchical summarization**: The system successfully processed repositories ranging from 4 to 574 functions while maintaining consistent quality (4.57/5.0 overall). Repository-level summaries scored highest (4.83/5.0) despite being the most abstract, showing that bottom-up aggregation preserves accuracy while managing complexity through strategic chunking (30 functions per module, 40 modules per repository).
    - **Strong inter-model agreement**: The 80% average agreement between Sonnet and Opus on dead code classification, with 100% agreement on clear-cut cases (unreachable code, empty functions), validates the system's reliability for unambiguous scenarios while appropriately flagging edge cases for human review.
- **Limitations**
    - **Single evaluator bias**: Using only Claude Opus for quality assessment introduces potential bias, both Sonnet and Opus share similar training data and architectural patterns. Human expert evaluation would provide more robust validation but at significantly higher cost.
    - **Python-only implementation**: RepoSpeak's AST parser is tightly coupled to Python syntax and semantics. Extending to other languages requires language-specific parsers and detection heuristics. Dynamically-typed languages like JavaScript present additional challenges for dead code detection without runtime analysis.
    - **Scalability ceiling**: Despite chunking strategies, repositories exceeding 10,000 functions would face context window challenges. The rich library (574 functions) required careful batching, larger monorepos might need more aggressive hierarchical grouping or directory-level summarization.
    - **Dead code detection gaps**: The system cannot detect reflection-based usage, dynamic imports, or framework conventions (e.g., Flask decorators creating implicit call paths). Therefore, certain patterns remain ambiguous without domain knowledge or runtime profiling.
- **Challenges**
    - **Handling API Reliability Issues**: During testing, Claude API overload errors frequently interrupted long-running analyses, particularly when processing large repositories like click as seen in Figure 16. Implementing exponential backoff retry logic (1s, 3s, 5s delays) was essential to prevent analysis failures and data loss. The challenge wasn't just adding retries, it required deciding which errors were transient (retry) versus permanent (abort), determining appropriate delay intervals to avoid overwhelming the API further, and maintaining analysis state across retries. This experience highlighted that production LLM applications must treat API calls as inherently unreliable and build resilience mechanisms from the start.

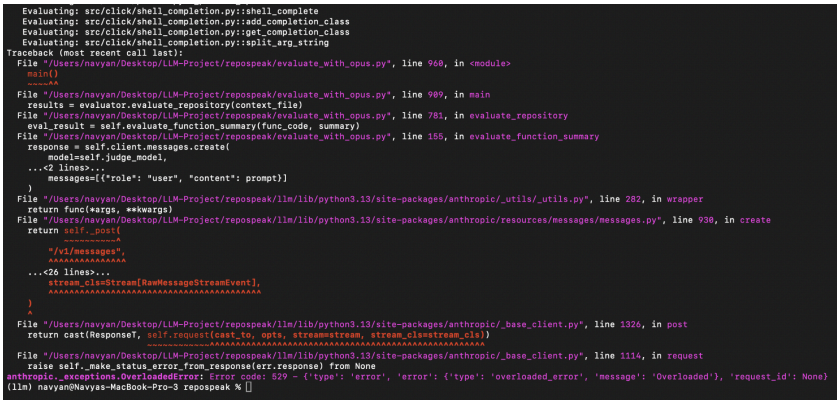


Figure 16: Claude API overload error

- **Balancing Completeness vs. Brevity**: The hardest part was crafting prompts that generated repository summaries with the right level of detail. Early attempts produced either overly generic descriptions ("a Python web framework") lacking actionable insights, or exhaustive technical catalogs that defeated summarization's purpose. For the rich library with 574 functions, learning to prompt for "high-level architecture with key technical decisions" rather than just "summarize this repository" required experimentation, the LLM needed explicit guidance on abstraction level rather than inferring appropriate granularity.

- **Handling Edge Cases in AST Parsing**: Python's dynamic nature creates challenges, decorators modify function behavior, metaclasses alter class instantiation. Learning when static analysis reaches its limits was crucial.

- **Future Work**

  - **Multi-language support**: through tree-sitter or similar polyglot parsers could extend RepoSpeak beyond Python while maintaining language-specific dead code heuristics tailored to each language's idioms and conventions.

  - **Interactive refinement workflows**: allowing developers to provide feedback on summaries or dead code classifications could create training data for improving future generations and fine-tuning detection rules based on project-specific patterns.

  - **MCP integration for persistent context**: The initial proposal included Model Context Protocol (MCP) integration for real-time context synchronization across development sessions. Implementing MCP would enable seamless IDE integration, allowing developers to query repository context directly from their editor and maintain synchronized documentation as code evolves, rather than relying on file-based JSON persistence.

  - **Empirical evaluation of dead code detection**: Testing against synthetic repositories with ground-truth labels and comparing with baseline tools like Vulture would quantify precision/recall improvements and validate whether semantic LLM validation justifies the additional costs over static-only analysis.

Conclusion

RepoSpeak shows that combining AST analysis with LLM semantic understanding can generate useful, multi-modal repository documentation. Testing on six Python repositories ranging from 4 to 574 functions, the system achieved 4.57/5.0 overall quality while producing technical summaries, architecture diagrams, audio narrations, and dead code reports.

The most important finding is that semantic validation solves a major problem with static analysis: false positives. While AST parsing flagged 1,926 potential dead code issues, LLM analysis revealed that 77.9% were actually legitimate code, library APIs and framework hooks that just weren't called internally. Two independent LLM evaluators agreed 80% of the time, suggesting this semantic approach is reliable for understanding code.

The hierarchical summarization approach scaled well through strategic chunking, processing up to 30 functions per module call and 40 modules per repository call. Repository summaries scored 4.83/5.0 despite condensing hundreds of functions into digestible overviews. Different output formats served different needs: technical summaries for developers, conversational audio for non-technical stakeholders, and visual diagrams for understanding architecture.

Processing medium-sized repositories takes 30-70 minutes, which is reasonable compared to the hours required for manual documentation. Current limitations include Python-only support, potential bias from using only Claude models for evaluation, and difficulty detecting code loaded dynamically at runtime. Future improvements could add multi-language support, incorporate runtime profiling data, and validate usefulness with actual developers.

This project demonstrates that LLMs paired with program analysis can meaningfully automate repository documentation tasks that traditionally consume significant developer time. The results suggest a future where comprehensive documentation generation becomes a natural part of the development workflow rather than an afterthought.

Code Setup and Execution

- **System Requirements**
    - Python: 3.8 or higher
    - Operating System: macOS, Linux, or Windows
    - API Access: Anthropic API key (Claude Sonnet 4.5 and Opus 4.5)
- **Installation**
    - Run the command:
      git clone https://github.com/navyapriyanandi-tamu/u136004813_final_project_LLM.git
    - Run the command: cd repospeak
    - Create virtual environment by running the below commands
        - python3 -m venv llm
        - source llm/bin/activate
    - Run the command: pip install -r requirements.txt
- **Configuration**
    - Run the command: export ANTHROPIC_API_KEY=<your_anthropic_api_key_here>
    - With the above command, the key only persists for the current terminal session. We will need to run it again if you open a new terminal.
    - We could also run the command: echo "ANTHROPIC_API_KEY=your_api_key_here" > .env
- **Run the analysis Pipeline**
    - For executing the full pipeline (Analysis + Web Dashboard), run the command: python3 repospeak.py /path/to/target/repository.
    - For only analysis, run the command: python3 analyze_any_repo.py /path/to/target/repository
    - For only dashboard (using existing results), run the command: python3 repospeak.py /path/to/target/repository --skip-analysis
    - For only dashboard without mentioning target repository, run the command: streamlit run streamlit_app.py
    - The dashboard can be accessed at http://localhost:8501

- **Running Quality Evaluation**
    - To evaluate outputs with Claude Opus 4.5, run the command: python3 evaluate_with_opus.py /path/to/target/repository

- **Generated Outputs**

  After analysis, the following files are created in the repository root:

    - Core Analysis Files:
        - context_<repo>.json - Complete analysis state (all summaries, metadata, diagrams)
        - ast_analysis_<repo>.json - AST metadata and dead code candidates
        - dead_code_analysis_<repo>.json - LLM-validated dead code results
        - summaries_<repo>.json - All hierarchical summaries

    - Multi-Modal Outputs:
        - audio/<repo>.mp3 - Audio narration file
        - audio/<repo>_script.txt - Audio transcript
        - diagrams/<repo>_diagram.mmd - Mermaid diagram
        - diagrams/<repo>_tree.txt - ASCII dependency tree

    - Evaluation Outputs:
        - evaluation_<repo>.json - Complete evaluation of the outputs generated by Sonnet (quality scores, agreement rates)

Learning

- **Technical Skills**: Gained practical experience combining AST parsing for structural analysis with LLM API integration for semantic understanding, including implementing chunking, exponential backoff retry logic to handle API rate limits and overload errors during large repository processing.

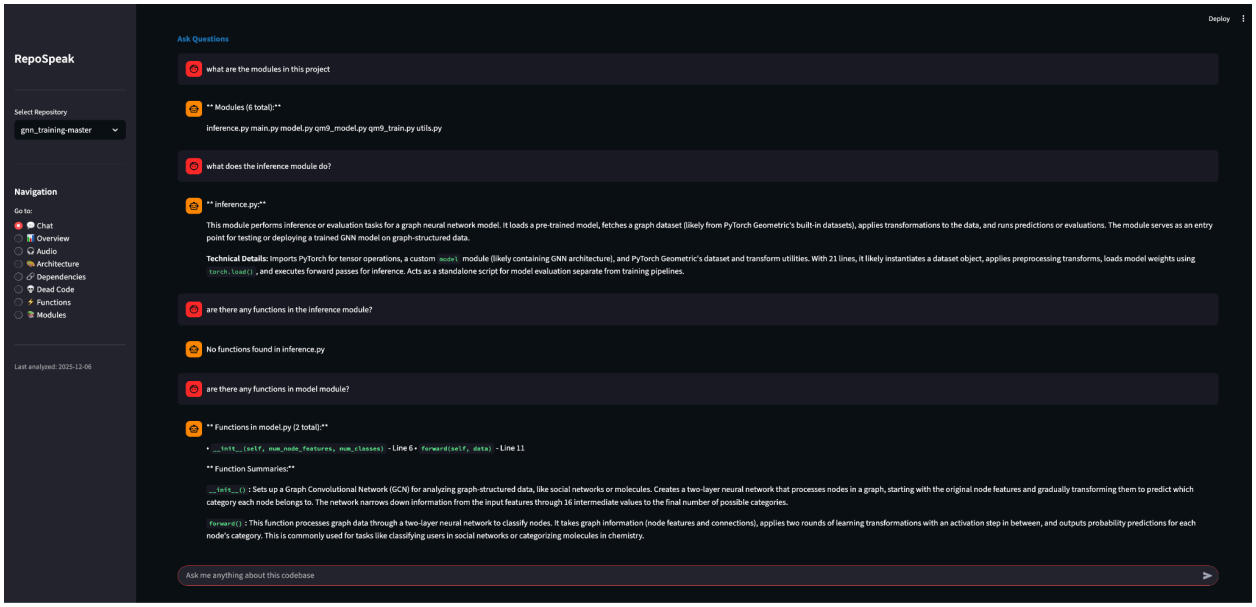
- **Conceptual Insights**: Learned that hybrid approaches address fundamental tradeoffs: static analysis provides comprehensive coverage but generates high false positives (77.9%), while LLMs add semantic validation but require carefully structured inputs and context management through hierarchical chunking strategies.

- **Development Practices**: Discovered that temperature parameters dramatically affect output characteristics, for example, using 0.2 for deterministic technical summaries versus 0.7 for creative conversational audio from the same model demonstrates how parameter tuning alone can serve different use cases.

References

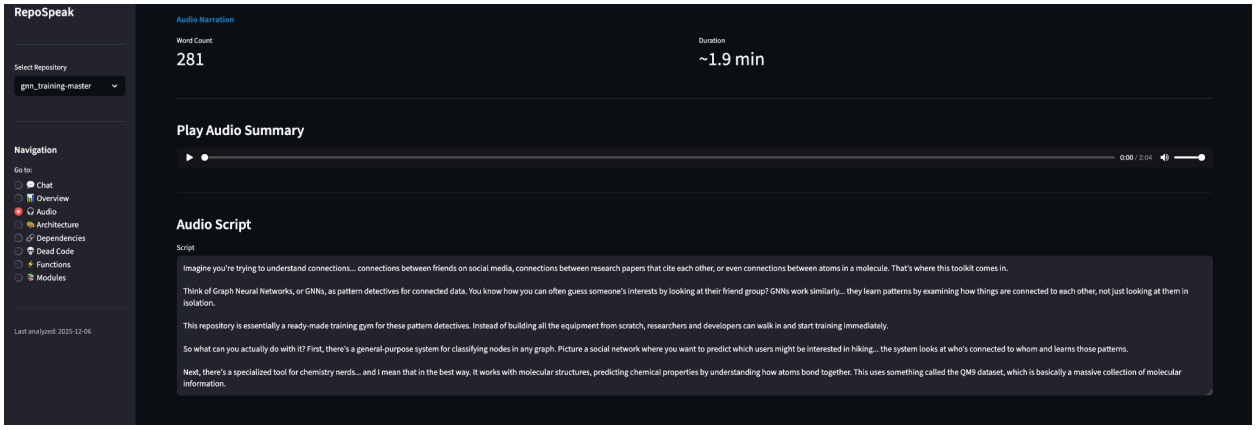
- Python Documentation, "ast - Abstract Syntax Trees," https://docs.python.org/3/library/ast.html
- L. Zheng et al., "Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena," NeurIPS 2023, 2023, https://arxiv.org/abs/2306.05685
- Y. Luo et al., "RepoAgent: An LLM-Powered Open-Source Framework for Repository-level Code Documentation Generation" 2024, https://arxiv.org/abs/2402.16667
- P. Diggs et al., "Leveraging LLMs for Legacy Code Modernization: Challenges and Opportunities for LLM-Generated Documentation," 2024, https://arxiv.org/html/2411.14971v1
- M. Li et al., "Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach" ACM Transactions on Software Engineering and Methodology, 2024, https://dl.acm.org/doi/10.1145/3649828
- M. Li et al., "IRIS: LLM-Assisted Static Analysis for Detecting Security Vulnerabilities" 2025, https://arxiv.org/abs/2405.17238

Screenshots of UI

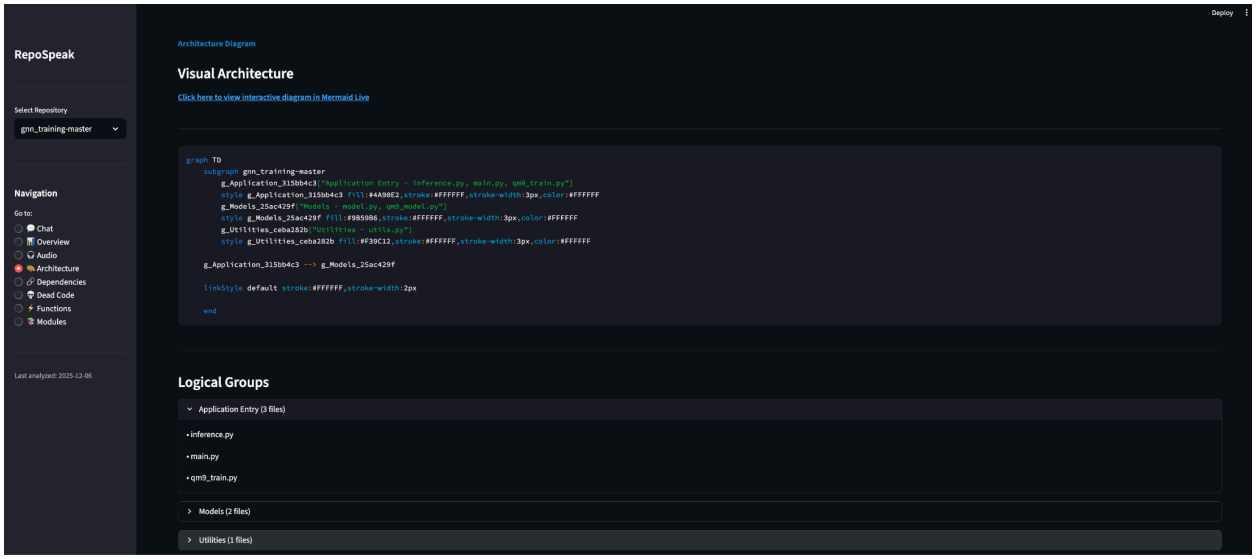


Screen 1: Interactive chat window

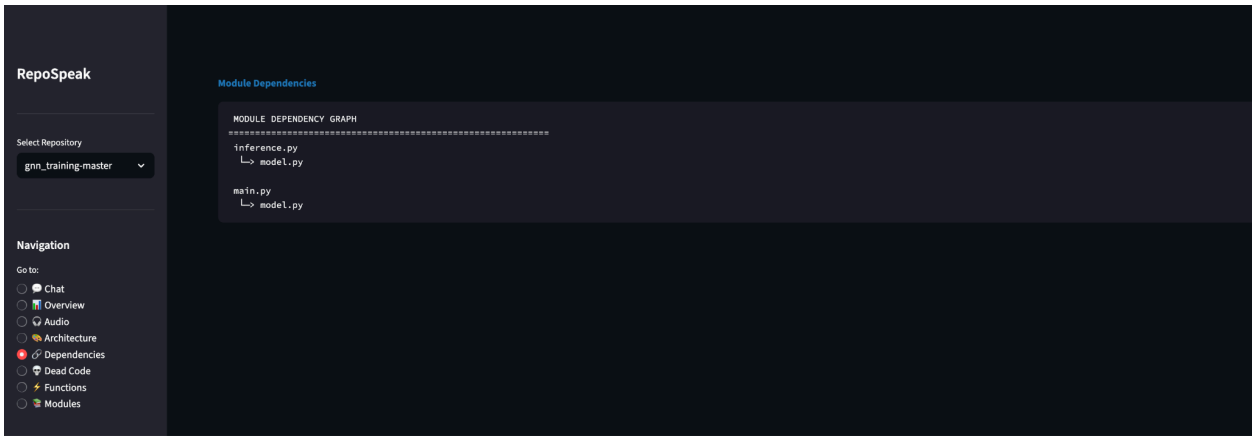
Screen 2: Repository Summary and overview



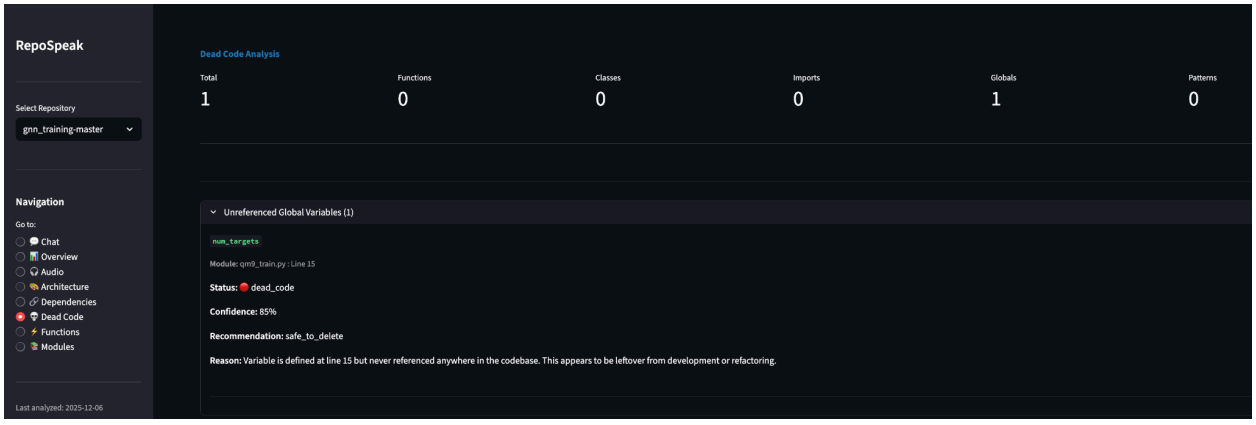
Screen 3: Audio Narration and transcript



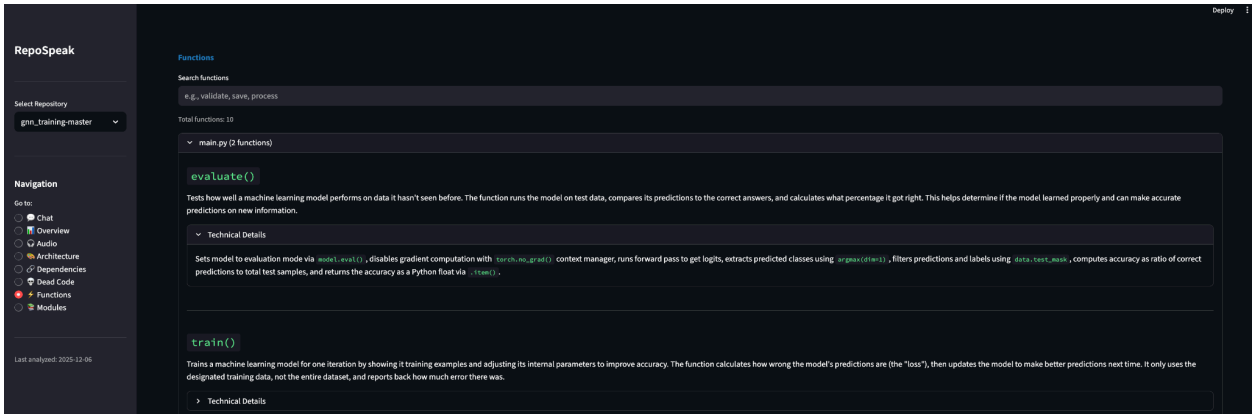
Screen 4: High Level architecture and Logical groups



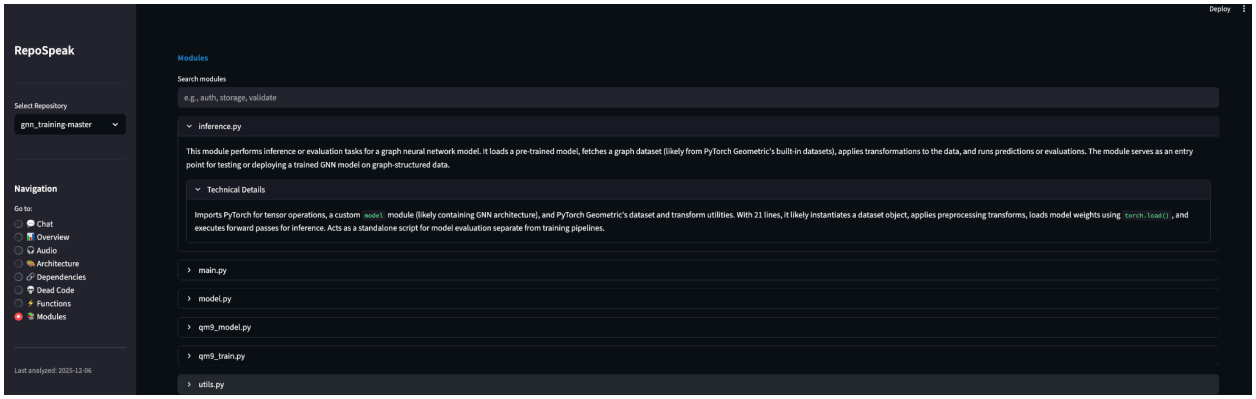
Screen 5: ASCII dependency Tree



Screen 6: Dead Code Analysis



Screen 7: Functions Summary



Screen 8: Modules Summary

```
(llm) navyan@Navyas-MacBook-Pro-3 repospeak % python3 evaluate_with_opus.py context_gnn_training-master.json

================================================================================
 Evaluating Repository Analysis with Claude Opus
================================================================================

 Repository: gnn_training-master

 Evaluating Repository Summary

 Evaluating Module Summaries
   Evaluating module: inference.py
   Evaluating module: main.py
   Evaluating module: model.py
   Evaluating module: qm9_model.py
   Evaluating module: qm9_train.py
   Evaluating module: utils.py

 Evaluating Function Summaries
   Evaluating: main.py::train
   Evaluating: main.py::evaluate
   Evaluating: model.py::__init__
   Evaluating: model.py::forward
   Evaluating: qm9_model.py::__init__
   Evaluating: qm9_model.py::forward
   Evaluating: qm9_train.py::train
   Evaluating: qm9_train.py::evaluate
   Evaluating: qm9_train.py::__init__
   Evaluating: qm9_train.py::forward

 Evaluating Audio Transcript

 Evaluating Architecture Diagram

 Evaluating Dead Code Detection
   Validating 1 dead code items

 Calculating Overall Metrics

================================================================================
 EVALUATION COMPLETE
================================================================================

 Overall Metrics:
   Repo Summary:       5.00/5.0
   Module Summaries:   4.17/5.0
   Function Summaries: 4.60/5.0
   Audio Transcript:   5.00/5.0
   Architecture:       4.00/5.0
   Dead Code Agreement:100.0% (Opus agrees with Sonnet)

     Status Agreement Breakdown (3-way classification):
       Dead Code: 1/1 agreements (100%)

     Category Breakdown:
       Unused Global Variables: 1/1 agreements (100%)

     Overall Quality:  4.63/5.0

 Full results saved to: evaluation_gnn_training-master.json
================================================================================
```

Screen 9: Terminal Output after Evaluation