SMARTBRIDGE
Let's Bridge the Gap

Code-to-Diagram Generator

Automated Code Visualization

Project Description

Manually producing diagrams for **documentation, code reviews, and developer onboarding** across complex codebases is slow, prone to errors, and often results in outdated visuals. The **Code-to-Diagram Generator** addresses this critical inefficiency by providing an **AI-assisted developer tool** that converts source code (Python, Java, Jupyter Notebooks) into accurate, maintainable visual representations.

By integrating **static code parsing** (parser_module.py), in-depth **structural analysis** (analyzer.py), and **LLM-based semantic interpretation** (llm_integration.py) with programmatic rendering technologies like Graphviz (diagram_generator.py), the tool establishes an end-to-end pipeline. The primary output includes industry-standard formats such as **UML Class Diagrams**, **Flowcharts**, and **Sequence Diagrams**, accelerating comprehension and reducing manual documentation time significantly. The final output is delivered via a lightweight Streamlit UI (app.py), allowing users to preview, edit, and export diagrams in formats like PNG, SVG, Markdown, and DOT source.

The tool focuses on **clarity and accuracy** of structural and logical components, providing developer-readable diagrams that capture major dependencies and control flows, making large or legacy codebases instantly navigable.

Scenario 1: Onboarding a New Engineer with Class Hierarchy Visualization

Users begin by uploading a **module or a Python file** into the tool via the file uploader. After the code is ingested, the system's **Parser Module** extracts classes, methods, and inheritance chains. The **Analyzer** determines the main components and relationships. The diagram_generator.py module then automatically produces a **UML Class Diagram** showing class relationships, inheritance (extends), and key methods for all defined objects.

The visualization is balanced across **detail and readability**, ensuring a new team member can grasp the core architectural pattern and component responsibilities within minutes of uploading the file.

Scenario 2: Code Review with AI-Based Flowchart Generation

In the **Main Analysis Page**, a reviewer uploads a newly modified Python script intended for a pull request. Based on the presence of conditional logic (if/else) and loops (for/while), the tool identifies the need for a control-flow visualization. The **LLM Interpreter** component is then used to summarize the overall purpose of the method and label the key flow components.

The AI allocates the visualization effort towards creating a clear, simple **Flowchart** (using the create_simple_diagram logic if too complex), highlighting the main decision points and execution path. This helps the reviewer quickly verify the new logic path against the design document without financial guesswork.

Scenario 3: Generating Documentation Artifacts with Export Capabilities

The Documentation Planner allows technical writers to process a Java code sample defining an API layer. The **Parser Module** extracts all public methods, return types, and fields, treating them as structural nodes. The **Analyzer** detects dependencies between internal service classes. The diagram_generator.py offers the UML diagram, but the technical writer needs a **Markdown artifact** for their API documentation.

The system analyzes the data and offers the image options, and upon clicking the **"Export All Formats"** button, the exporter.py module renders the Graphviz source to PNG/SVG and packages the visual into a .md file, embedding the image and the DOT source code. This ensures that the documentation remains elegant, matching the code structure, and is easy to update during refactoring.

SMARTBRIDGE
Let's Bridge the Gap

Architecture Overview

The **Code-to-Diagram Generator** is a modular, AI-assisted development tool designed to convert source code into clear, maintainable visual representations. It utilizes a **Python pipeline** built around Streamlit for the UI, static analysis libraries for parsing, and the Groq API for fast LLM-based interpretation, ensuring rapid visualization of code structure and logic.

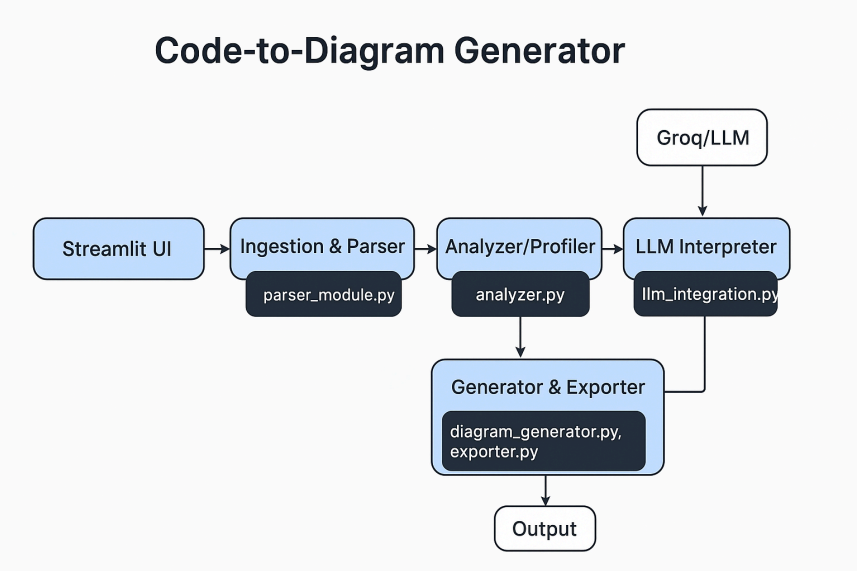
Core Technologies

- **Streamlit (app.py, viewer.py):** Serves as the **lightweight frontend** and handles the application's overall **state management** and visualization display.
- **Graphviz (diagram_generator.py, exporter.py):** The primary **rendering engine** responsible for programmatically generating diagram definitions (DOT language) and exporting them to raster/vector image formats (PNG, SVG).
- **Groq/LLM (llm_integration.py):** The AI layer used for **semantic interpretation**, resolving ambiguous code intent, summarizing high-level relationships, and answering user queries about the analyzed code structure.
- **Parsing Libraries (parser_module.py):** Core static analysis tools (e.g., Python's ast, javalang, nbformat) used to reliably extract structural information and build the Abstract Syntax Tree (AST).

Component-Wise Architecture

Component	Description
Streamlit UI (app.py, styles.css)	The user-facing application provides drag-and-drop file upload, project navigation (Main Analysis, Code Queries), **interactive diagram preview/editing**, and final export controls.
Ingestion & Parser (parser_module.py)	Processes uploaded files (.py, .java, .ipynb). Builds the **Abstract Syntax Tree (AST)** and extracts raw structural data: classes, methods, functions, inheritance bases, and call sites.

Analyzer/Profiler (analyzer.py)	Performs static analysis on the parsed structure to map relationships: identifying **call graphs**, inheritance hierarchy, import dependencies, and calculating complexity metrics.
LLM Interpreter (llm_integration.py)	Takes the analyzed structure/diagram code and provides **natural language explanations**, summarizes code patterns, and handles direct, context-aware user queries about the architecture.
Generator & Exporter (diagram_generator.py, exporter.py)	The core transformation engine. Converts the structured analysis output into **Graphviz DOT source code** and renders it into final shareable formats (PNG, SVG, Markdown, DOT).



Pre-requisites

1. **Python Environment:** Python 3.8+ for Streamlit and dependencies.
2. **Graphviz Installation:** The Graphviz binaries must be installed and accessible via the system PATH for the image rendering to work (dot executable).
3. **Groq API Key:** A valid **GROQ_API_KEY** is required in the local .env file for all AI-assisted features (explanations, chat queries).
4. **Java Parsing Dependency:** The javalang package requires a correctly configured environment if Java code is to be parsed.

SMARTBRIDGE
Let's Bridge the Gap

Project Workflow

1. Core Tool Setup & Initialization

- **Activity 1.1:** Set up local environment: Install **Graphviz binaries** and verify system PATH access for rendering images (PNG/SVG).
- **Activity 1.2:** Configure **LLM API access** by setting the GROQ_API_KEY in the .env file and confirming API availability in llm_integration.py.
- **Activity 1.3:** Design the core structured **JSON schema** for the parsed data and the analyzed data, ensuring seamless transition between parser_module.py and analyzer.py.

2. Core Functionality Development (Parsing & Analysis)

- **Activity 2.1:** Develop the **parser_module.py** class structure to handle file ingestion and perform AST extraction for Python (ast) and Java (javalang), generating a raw structural summary.
- **Activity 2.2:** Implement the **analyzer.py** logic to transform raw parsed data into relationship maps, including: **call graph edges**, class inheritance, import categorization, and initial complexity assessment.
- **Activity 2.3:** Integrate the **parser_module.py** and **analyzer.py** outputs into the Streamlit session state in app.py after a successful file upload.

3. Diagram Generation & Exporter Implementation

- **Activity 3.1:** Define the UML and flow diagram generation functions in **diagram_generator.py**, focusing on converting the **analyzed relationships** (inheritance, calls) into the **Graphviz DOT language** and applying custom styling (theme_config.py).
- **Activity 3.2:** Develop the **exporter.py** class to reliably take the DOT source code and render it to multiple output formats: PNG, SVG, DOT, and Markdown (embedding the generated PNG).
- **Activity 3.3:** Create the basic error diagram functions (create_error_diagram) to handle and display Graphviz syntax issues or runtime errors gracefully.

4. UI/LLM Integration Development

- **Activity 4.1:** Build the main **Streamlit application (app.py)** for file upload, navigation (Main Analysis, Code Queries), and display routing.
- **Activity 4.2:** Implement the Diagram Editor/Viewer (viewer.py) functionality, allowing users to **preview changes** to the Graphviz source and apply/reset the modified diagram.

- **Activity 4.3:** Integrate the Groq LLM calls (llm_integration.py) to automatically generate **diagram explanations** on the Main Analysis page and enable the **real-time chat interface** on the Code Queries page.

5. Testing & Optimization

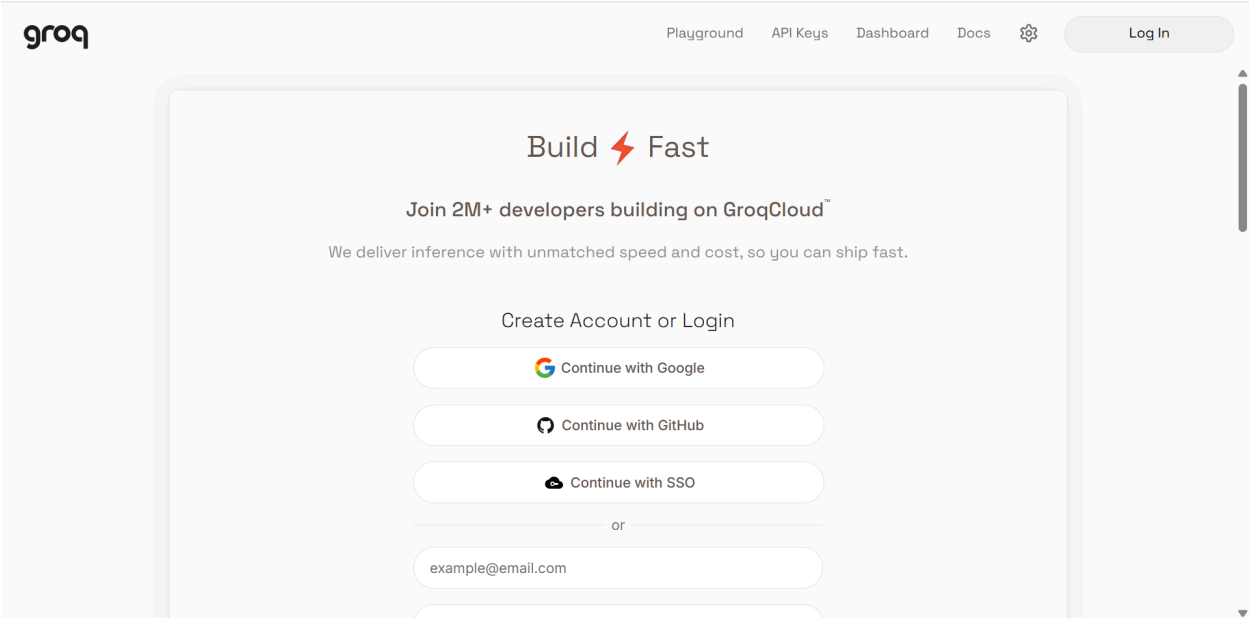
- **Activity 5.1:** Conduct end-to-end testing using complex Python, simple Python, and Java files to validate parser accuracy and call graph generation.
- **Activity 5.2:** Evaluate the stability of Graphviz rendering, especially for very large codebases, and optimize diagram layouts (e.g., using create_simple_diagram checkbox).
- **Activity 5.3:** Optimize LLM prompts to ensure concise, technical, and high-quality responses for both code queries and diagram summaries.
- **Activity 5.4:** Validate the full export pipeline (export_all_four_formats), ensuring all file types are generated correctly and download smoothly from the Streamlit UI.

MILESTONE 1: LLM Infrastructure Setup (Groq Integration)

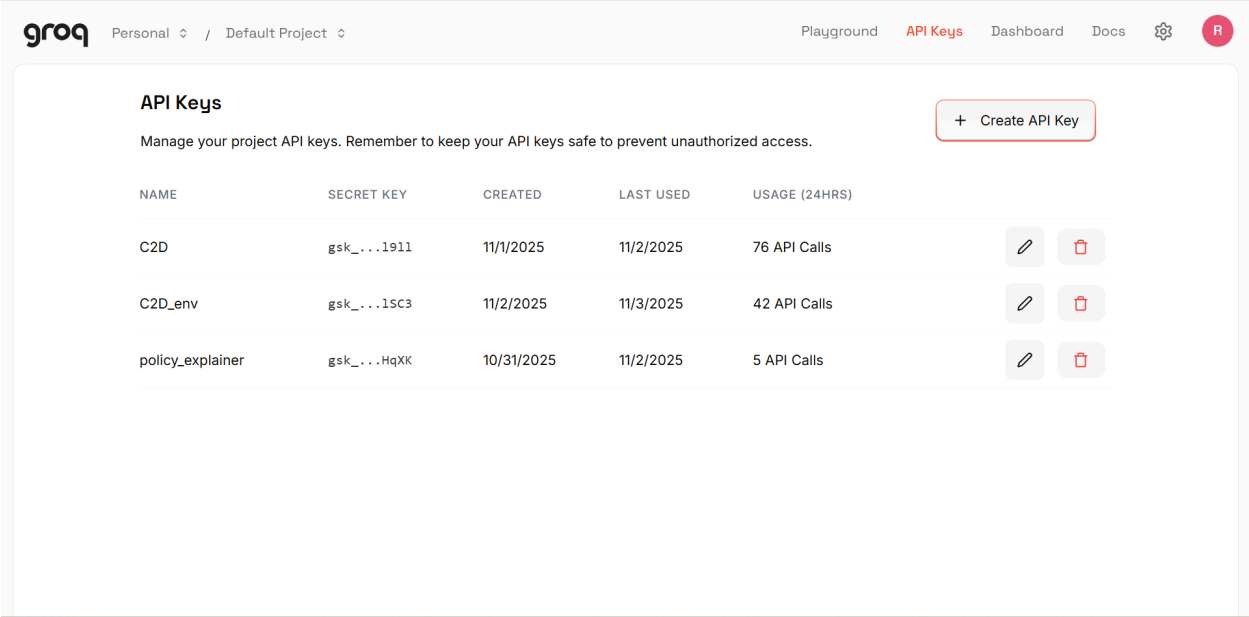
In this foundational stage, the objective is to establish and configure the core **LLM infrastructure** required to power the analysis and query functionalities of the Code-to-Diagram Generator. The platform relies on fast, high-quality models (like Llama 3) accessed via the **Groq API** to provide rapid code summaries, detailed diagram explanations, and answer user queries. This milestone ensures the development environment is equipped with authenticated access to Groq's APIs, enabling all AI-driven functionalities—such as semantic interpretation and chat interaction—to operate seamlessly.

Activity 1.1: Set up Groq Cloud account and obtain API Key

- Sign in or create a Groq Cloud account via the official **Groq Console**.



- Navigate to the **API Keys** section within the dashboard.



- Click on **"Create API Key"** and name the key (e.g., CodeDiagramGenerator).
- **Copy the generated API Key** immediately, as it will not be shown again.

Activity 1.2: Configure access in the Development Environment

- In your project directory, locate the **.env** file.

- **Rationale:** The llm_integration.py file uses python-dotenv to securely load this key, preventing sensitive credentials from being committed to source control.

Activity 1.3: Validate Groq API connectivity and model configuration

- Confirm that the base URL and model name (llama-3.3-70b-versatile or preferred Groq model) are correctly configured in **llm_integration.py**.

```python
class LLMIntegration:
    """Simple LLM integration for code analysis"""

    def __init__(self):
        self.api_key = os.getenv("GROQ_API_KEY")
        self.model = "llama-3.3-70b-versatile"
        self.base_url = "https://api.groq.com/openai/v1/chat/completions"

```

- Run a quick test function (outside of Streamlit) using the requests library to confirm the key works and returns a response.
- The function **is_api_available()** in llm_integration.py should return True when run, confirming the key is loaded and environment access is valid.
- **Focus:** Ensure the system prompt for the llm_integration.py calls is effective in generating concise, technical code analysis responses.

MILESTONE 2: Core Pipeline Development (Parsing, Analysis & Generation)

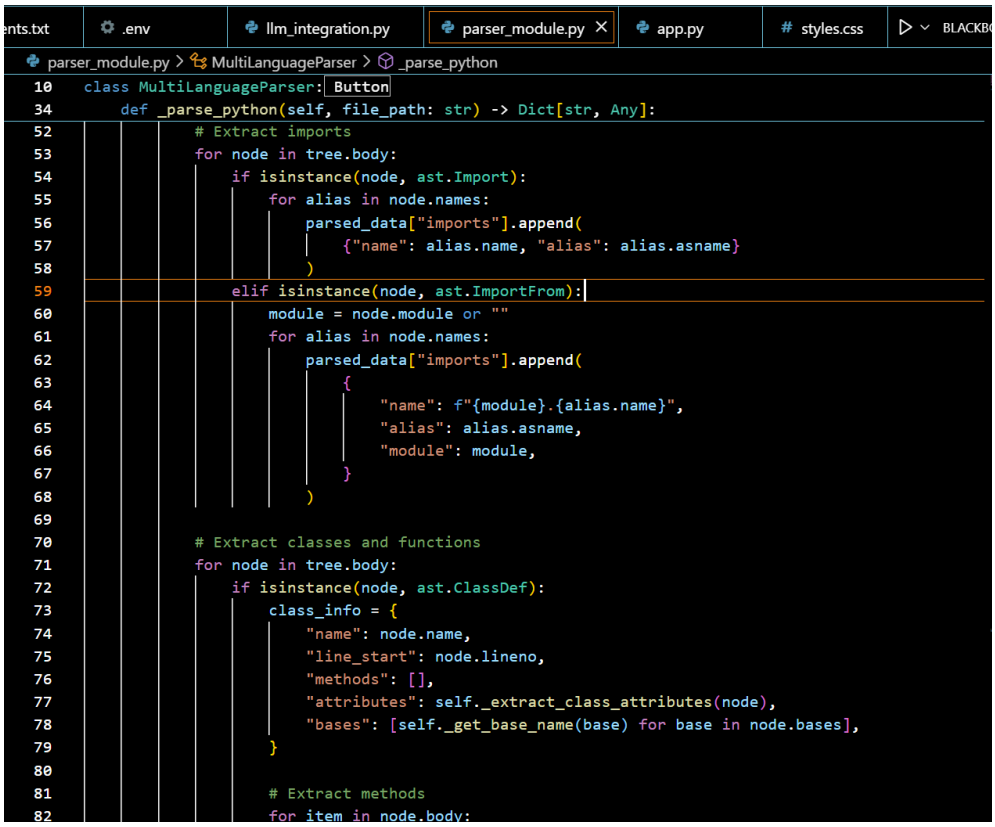
This milestone focuses on building the core structural and relational intelligence of the Code-to-Diagram Generator, encompassing the first three stages of the pipeline: **Parsing (AST Extraction), Static Analysis (Relationship Mapping), and Diagram Generation (DOT Source)**. The primary goal is to develop a robust, modular system capable of ingesting code, extracting key components, inferring their relationships, and translating this structure into a programmatic diagram definition. All core data transformation logic is contained within dedicated utility modules (parser_module.py, analyzer.py, diagram_generator.py).

File Explorer Structure

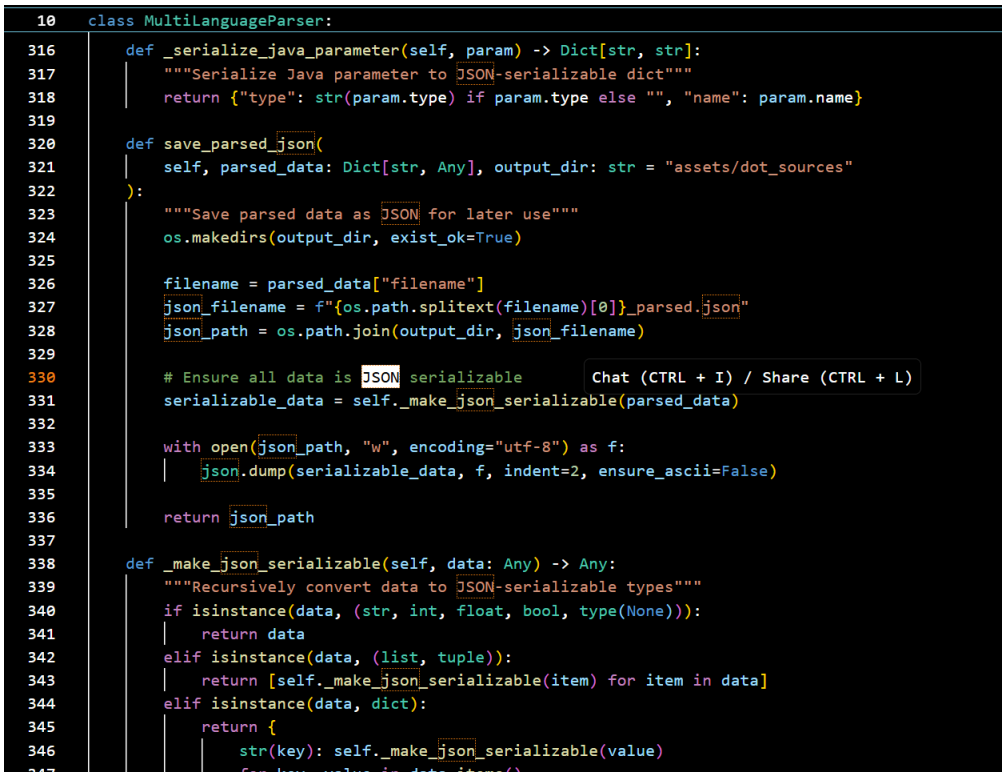
Activity 2.1: Parser Initialization and Structural Extraction (parser_module.py)

- **Implement Multi-Language Parser:** Finalize the MultiLanguageParser class to correctly handle Python (ast), Java (javalang), and Jupyter Notebook (nbformat) file types.

- **Structural Extraction:** Ensure the parser reliably extracts **all key structural nodes** as detailed in the project description: classes, methods, functions, inheritance bases (bases in Python, extends/implements in Java), and all internal function/method calls.



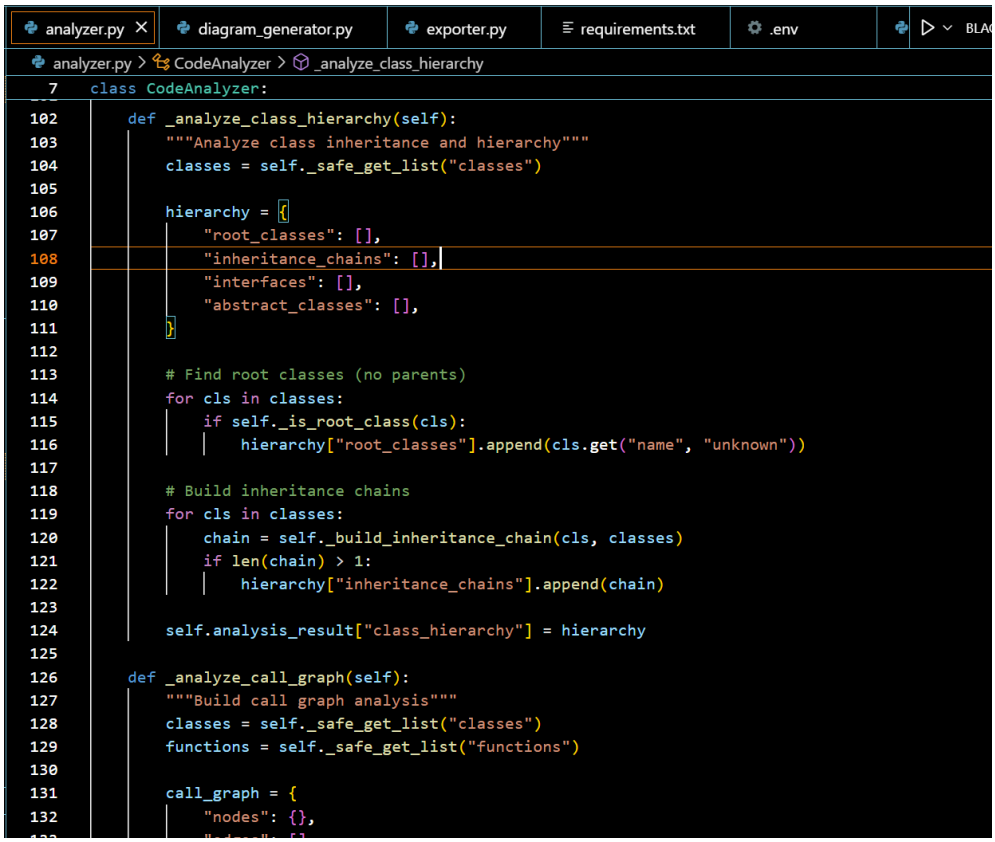
- **Data Standardization:** Guarantee the output structure (the parsed JSON dictionary) is uniform across languages to facilitate generic consumption by the analyzer.py.



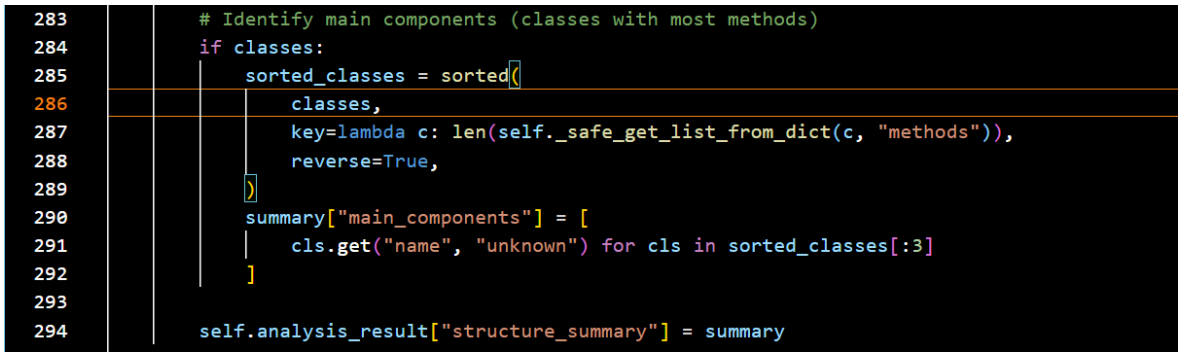
```
class MultiLanguageParser:
    def _serialize_java_parameter(self, param) -> Dict[str, str]:
        """Serialize Java parameter to JSON-serializable dict"""
        return {"type": str(param.type) if param.type else "", "name": param.name}

    def save_parsed_json(
        self, parsed_data: Dict[str, Any], output_dir: str = "assets/dot_sources"
    ):
        """Save parsed data as JSON for later use"""
        os.makedirs(output_dir, exist_ok=True)

        filename = parsed_data["filename"]
        json_filename = f"{os.path.splitext(filename)[0]}_parsed.json"
        json_path = os.path.join(output_dir, json_filename)

        # Ensure all data is JSON serializable        Chat (CTRL + I) / Share (CTRL + L)
        serializable_data = self._make_json_serializable(parsed_data)

        with open(json_path, "w", encoding="utf-8") as f:
            json.dump(serializable_data, f, indent=2, ensure_ascii=False)

        return json_path

    def _make_json_serializable(self, data: Any) -> Any:
        """Recursively convert data to JSON-serializable types"""
        if isinstance(data, (str, int, float, bool, type(None))):
            return data
        elif isinstance(data, (list, tuple)):
            return [self._make_json_serializable(item) for item in data]
        elif isinstance(data, dict):
            return {
                str(key): self._make_json_serializable(value)
```

Activity 2.2: Static Analysis and Relationship Mapping (analyzer.py)

- **Build Relationships:** Implement the CodeAnalyzer methods to process the raw parsed data and generate **high-level relationship maps**.
    - /analyze_class_hierarchy: Maps class inheritance chains and interfaces.
    - /analyze_call_graph: Determines function/method invocation edges across the module.

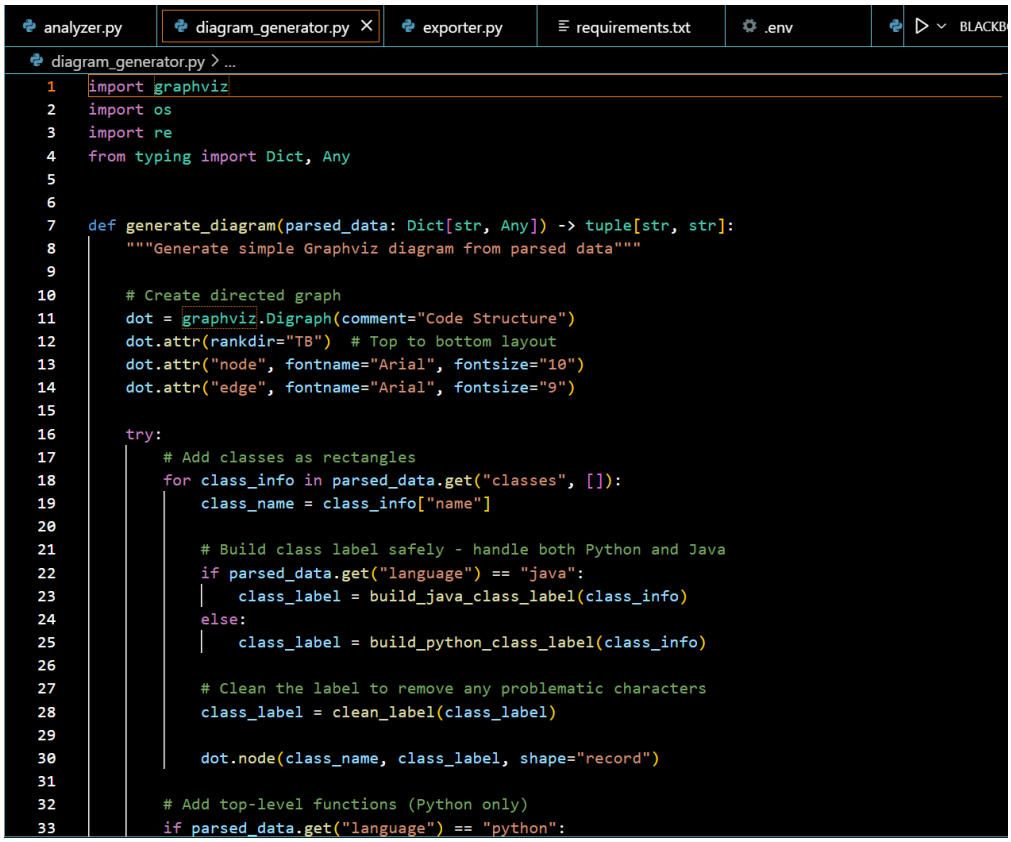


- **Contextual Metrics:** Implement and refine the logic to calculate **complexity metrics** and determine the **structure_summary** (e.g., "Object-Oriented," "Procedural"), which will inform the LLM and the main UI summary.

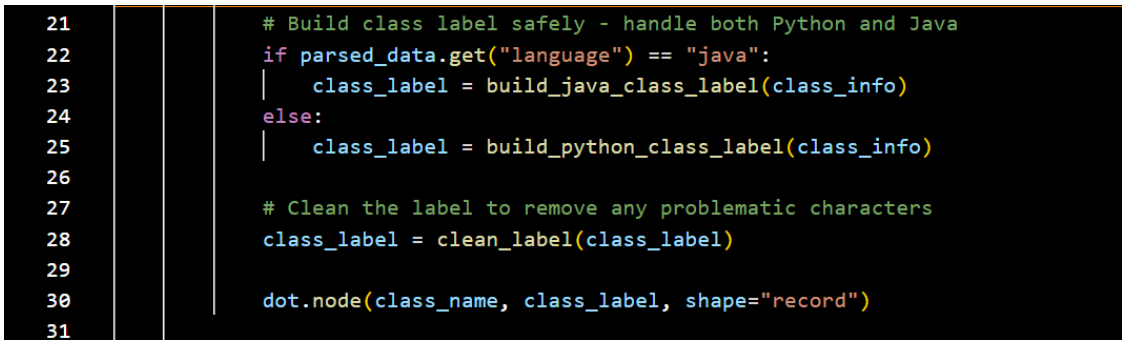


Activity 2.3: Diagram Source Generation (diagram_generator.py)

- **DOT Source Conversion:** Implement the generate_diagram function to take the **analyzed data structure** and translate it into a valid **Graphviz DOT source code** string.



- **UML Label Formatting:** Develop the helper functions (build_java_class_label, build_python_class_label) to create well-structured **UML record labels** displaying key attributes, fields, and methods with appropriate formatting.



- **Simple Diagram Fallback:** Finalize the create_simple_diagram function, ensuring it provides a basic overview of high-level components (classes/functions) for complex or large files where a full diagram may be overwhelming.


```python
def create_simple_diagram(parsed_data: Dict[str, Any]) -> tuple[str, str]:
    """Alternative simple diagram generator for complex files"""
    dot = graphviz.Digraph(comment="Simple Code Structure")
    dot.attr(rankdir="TB")
    dot.attr("node", shape="box", style="rounded", fontname="Arial")

    # Just show classes and basic info
    for class_info in parsed_data.get("classes", []):
        class_name = class_info["name"]
        method_count = len(class_info.get("methods", []))
        field_count = len(class_info.get("fields", []))
        attr_count = len(class_info.get("attributes", []))

        if parsed_data.get("language") == "java":
            label = f"{class_name}\\n{method_count} methods\\n{field_count} fields"
        else:
            label = f"{class_name}\\n{method_count} methods\\n{attr_count} attributes"

        dot.node(class_name, clean_label(label))

    # Add inheritance
    if parsed_data.get("language") == "java":
        for class_info in parsed_data.get("classes", []):
            if class_info.get("extends"):
                dot.edge(
                    class_info["name"],
                    class_info["extends"],
                    style="dashed",
                    label="extends",
                )

    # Add function calls for Python
```

Activity 2.4: Integration and Data Flow

- **Pipeline Assembly (app.py):** Ensure the process_uploaded_file function in app.py correctly orchestrates the sequence: **Parser** → **Analyzer**.

.env | llm_integration.py | parser_module.py | app.py | styles.css | the | BLACKB

app.py > process_uploaded_file

```python
def process_uploaded_file(uploaded_file):
    """Process the uploaded file through the entire pipeline"""
    file_ext = os.path.splitext(uploaded_file.name)[1].lower()

    with tempfile.NamedTemporaryFile(delete=False, suffix=file_ext) as tmp_file:
        tmp_file.write(uploaded_file.getvalue())
        temp_path = tmp_file.name

    try:
        # Read file content
        with open(temp_path, "r", encoding="utf-8") as f:
            st.session_state.source_code_content = f.read()

        # Parse and analyze code
        with st.spinner("Parsing code structure..."):
            parsed_data = parse_file(temp_path, save_json=True)

        with st.spinner("Analyzing code relationships..."):
            analyzed_data = analyze_code(parsed_data, save_json=True)

        # Update session state
        update_session_after_processing(analyzed_data, uploaded_file)

        st.success("Analysis complete!")

    except Exception as e:
        st.error(f"Error processing file: {str(e)}")
    finally:
        # Clean up temporary file
        if os.path.exists(temp_path):
            os.unlink(temp_path)
```

- **Input-Output Loop:** Verify that the output dictionary from analyzer.py is successfully stored in st.session_state.processed_data and contains all the necessary data points required by **diagram_generator.py** to begin visualization.

```python
def update_session_after_processing(analyzed_data, uploaded_file):
    """Update session state after processing file"""
    st.session_state.processed_data = analyzed_data
    st.session_state.current_file = uploaded_file
    st.session_state.query_page_chat_history = []
    st.session_state.last_api_call_time = 0
    st.session_state.edited_graphviz = ""
    st.session_state.diagram_updated = False
    st.session_state.last_error = ""
    st.session_state.exported_files = {}
    st.session_state.diagram_explanation = ""
    st.session_state.last_button_press_time = time.time()
    st.session_state.last_processed_question = None
```

SMARTBRIDGE
Let's Bridge the Gap

MILESTONE 3: UI & LLM Bridge Implementation (app.py, llm_integration.py)

This milestone focuses on implementing the **Streamlit-powered user interface** (app.py) and the **LLM interaction layer** (llm_integration.py). This combination serves as the critical bridge, taking the analyzed code structure and presenting it to the user through visualizations and AI-driven insights. It is responsible for routing between the core analysis pipeline and the interactive code query tool.

Activity 3.1: Define Streamlit Page Routing

- **Implement Main Analysis Page (main_analysis_page):** Serves as the primary entry point for file upload, triggers the **Parse** → **Analyze** → **Generate Diagram** pipeline, and displays the static results (metrics, diagram preview, LLM explanation).

```python
def main_analysis_page():
    """Main analysis page with file upload and diagram generation"""

    # File upload section
    uploaded_file = st.file_uploader(
        "Choose a code file",
        type=["py", "java", "ipynb"],
        help="Supported formats: Python (.py), Java (.java), Jupyter Notebook (.ipynb)",
        label_visibility="collapsed",
    )

    # Process uploaded file
    if uploaded_file and uploaded_file != st.session_state.current_file:
        process_uploaded_file(uploaded_file)

    # Display results if data is available
    if st.session_state.processed_data:
        display_analysis_results()
        display_diagram_section()
        display_diagram_explanation()
        display_export_section()
```

- **Implement Code Queries Page (code_queries_page):** Acts as the **interactive chat interface**, displaying the current diagram and managing the conversation history, enabling users to ask detailed questions about the code structure.

```python
def code_queries_page():
    """Separate page for code queries with diagram and chat"""
    if not st.session_state.processed_data:
        st.warning(
            "Please upload and analyze a code file first from the Main Analysis page."
        )
        return

    # API status
    if not is_api_available():
        st.error("GROQ_API_KEY not found in .env file. AI features require API key.")

    # Single column layout for diagram only
    display_diagram_viewer_section()

    # Chat interface
    display_query_chat_interface()

```

- **Navigation Setup:** Ensure the sidebar radio buttons in app.py (which handle internal routing) reliably switch between the two main pages while preserving the application state (e.g., the analyzed data).

```python
def main():
    """Main application function"""
    initialize_session_state()
    reset_button_states()  # Reset button states on every run

    # Sidebar navigation
    st.sidebar.markdown(
        '<div class="sidebar-header">Navigation</div>', unsafe_allow_html=True
    )

    # Page selection
    page_options = ["Main Analysis", "Code Queries"]
    selected_page = st.sidebar.radio("", page_options, label_visibility="collapsed")

    # Update page state
    if selected_page != st.session_state.current_page:
        st.session_state.current_page = selected_page
        st.rerun()

    # Display page header
    display_page_header()

    # Route to appropriate page
    if st.session_state.current_page == "Main Analysis":
        main_analysis_page()
    else:
        code_queries_page()

```

Activity 3.2: LLM Integration and Explanation Services

- **Diagram Explanation (get_detailed_diagram_explanation):** Implement the logic in app.py to trigger the LLM via llm_integration.py immediately after diagram generation, providing a **concise, high-level summary** of the visualized architecture.

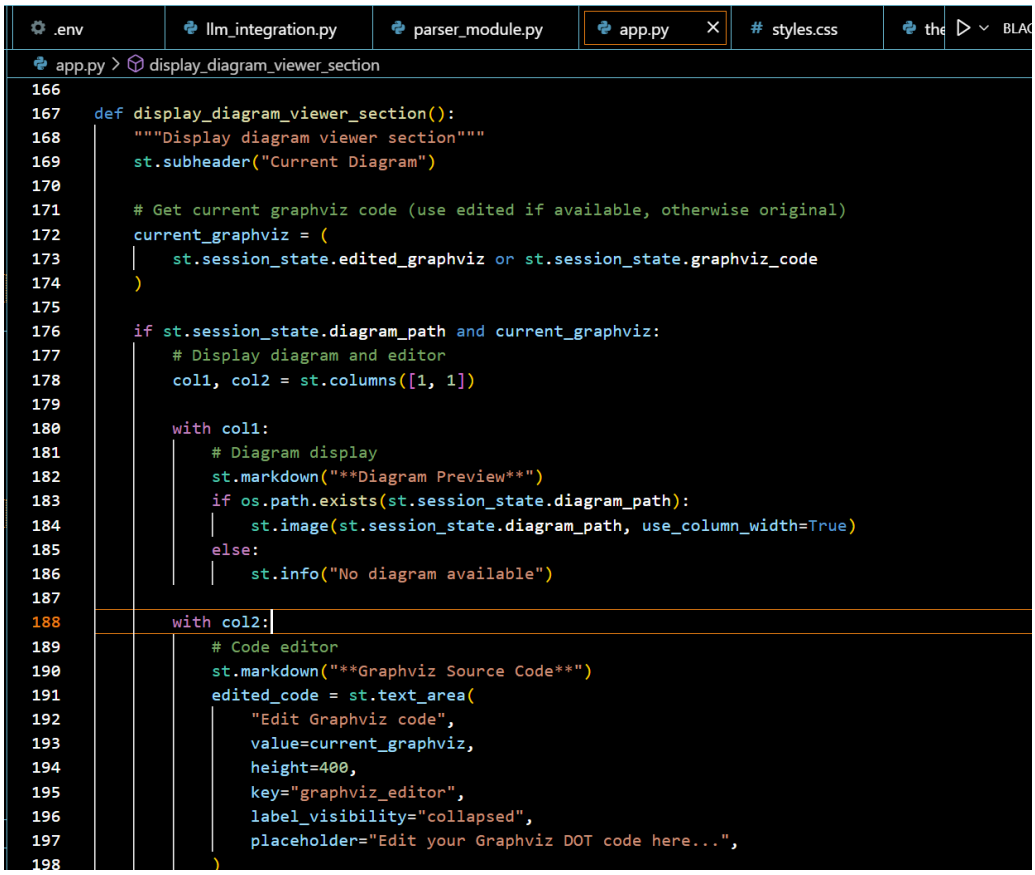
```python
# Public functions
def get_detailed_diagram_explanation(
    code_structure: Dict[str, Any], graphviz_code: str
) -> str:
    """Get 3-4 line diagram explanation"""
    return llm.get_diagram_explanation(code_structure, graphviz_code)


def get_code_analysis_response(
    question: str, source_code: str, code_structure: Dict[str, Any]
) -> str:
    """Get 3-4 line code analysis response"""
    return llm.get_code_response(question, source_code, code_structure)

```

- **Interactive Query Handler (process_question):** Design the handler in app.py to securely package the user's question, the source code, and the analyzed structure, passing it to llm_integration.py to generate a targeted response.

```python
def process_question(question):
    """Process a single question"""
    current_time = time.time()

    # Rate limiting: 1 second between requests
    if current_time - st.session_state.last_api_call_time < 1:
        return

    st.session_state.last_api_call_time = current_time

    # Add user question to history
    st.session_state.query_page_chat_history.append((question, True))

    # Get AI response
    with st.spinner("Analyzing your code..."):
        ai_response = get_code_analysis_response(
            question,
            st.session_state.source_code_content,
            st.session_state.processed_data,
        )

        # Add AI response to history
        st.session_state.query_page_chat_history.append((ai_response, False))

```

Activity 3.3: Diagram Viewer and Editing Controls

- **Editable Viewer (display_diagram_viewer_section in app.py):** Integrate the viewing and editing components from viewer.py into app.py, ensuring users can see the generated image alongside the editable **Graphviz DOT source code**.



- **State Persistence:** Implement the button logic for **"Apply Changes"** and **"Reset to Original"**, updating st.session_state.edited_graphviz and triggering the regenerate_diagram_from_edited_code function to reflect real-time visualization changes.

```
620
621 def regenerate_diagram_from_edited_code():
622     """Regenerate diagram from edited Graphviz code"""
623     if not st.session_state.edited_graphviz:
624         return False
625
626     try:
627         base_name = os.path.splitext(st.session_state.processed_data["filename"])[0]
628         diagram_path = f"assets/diagrams/{base_name}_edited"
629
630         dot = graphviz.Source(st.session_state.edited_graphviz)
631         dot.render(diagram_path, format="png", cleanup=True)
632
633         st.session_state.diagram_path = diagram_path + ".png"
634         st.session_state.diagram_updated = True
635         st.session_state.last_error = ""
636         st.session_state.diagram_explanation = ""
637         return True
638
639     except graphviz.ExecutableNotFound:
640         st.session_state.last_error = (
641             "Graphviz executable not found. Please install Graphviz."
642         )
643         return False
644     except graphviz.backend.CalledProcessError as e:
645         error_msg = str(e)
646         if "stderr: b'" in error_msg:
647             match = re.search(r"stderr: b'([^']*)'", error_msg)
648             if match:
649                 clean_error = (
650                     match.group(1).replace("\\r\\n", "\n").replace("\\n", "\n")
651                 )
652                 st.session_state.last_error = f"Graphviz syntax error:\n{clean_error}"
```

Activity 3.4: Startup and Initialization

- **State Initialization:** Finalize the initialize_session_state function in app.py to correctly set default values for all state variables (processed_data, graphviz_code, chat history, etc.) on application startup.

```python
def initialize_session_state():
    """Initialize all session state variables"""
    session_vars = {
        "processed_data": None,
        "current_file": None,
        "graphviz_code": None,
        "diagram_path": None,
        "source_code_content": "",
        "query_page_chat_history": [],
        "last_api_call_time": 0,
        "edited_graphviz": "",
        "diagram_updated": False,
        "last_error": "",
        "exported_files": {},
        "apply_changes_clicked": False,
        "reset_original_clicked": False,
        "current_page": "Main Analysis",
        "diagram_explanation": "",
        "last_button_press_time": 0,
        "last_processed_question": None,
    }

    for var, default in session_vars.items():
        if var not in st.session_state:
            st.session_state[var] = default
```

- **Dependency Check:** Ensure app.py includes the necessary import of is_api_available() to provide early warning to the user if the Groq API key is missing.

```python
    # API status
    if not is_api_available():
        st.error("GROQ_API_KEY not found in .env file. AI features require API key.")

    # Single column layout for diagram only
    display_diagram_viewer_section()

    # Chat interface
    display_query_chat_interface()
```

MILESTONE 4: UI Development & Design

This milestone focuses on developing the visual frontend and user experience for the Code-to-Diagram Generator, utilizing **Streamlit** for rapid development and **custom CSS** (styles.css, based on theme_config.py) for a responsive, modern look. The interface is designed to facilitate seamless user interaction through file upload, page navigation, and diagram editing.

Activity 4.1: Develop Streamlit Frontend Interface

- **Design Main Analysis Page (main_analysis_page):** Implements the drag-and-drop file uploader (for .py, .java, .ipynb), displays the **Code Analysis Summary** metrics in a card-like layout, and presents the generated diagram for preview.

```python
def main_analysis_page():
    """Main analysis page with file upload and diagram generation"""

    # File upload section
    uploaded_file = st.file_uploader(
        "Choose a code file",
        type=["py", "java", "ipynb"],
        help="Supported formats: Python (.py), Java (.java), Jupyter Notebook (.ipynb)",
        label_visibility="collapsed",
    )

    # Process uploaded file
    if uploaded_file and uploaded_file != st.session_state.current_file:
        process_uploaded_file(uploaded_file)

    # Display results if data is available
    if st.session_state.processed_data:
        display_analysis_results()
        display_diagram_section()
        display_diagram_explanation()
        display_export_section()


def code_queries_page():
    """Separate page for code queries with diagram and chat"""
    if not st.session_state.processed_data:
        st.warning(
            "Please upload and analyze a code file first from the Main Analysis page."
        )
        return

    # API status
```

- **Implement Code Queries Chat Interface (code_queries_page):** Designs the chat history display (chat-ai, chat-user CSS classes), the quick question buttons, and the text area for user input, ensuring a responsive chat UX.

```python
def code_queries_page():
    """Separate page for code queries with diagram and chat"""
    if not st.session_state.processed_data:
        st.warning(
            "Please upload and analyze a code file first from the Main Analysis page."
        )
        return

    # API status
    if not is_api_available():
        st.error("GROQ_API_KEY not found in .env file. AI features require API key.")

    # Single column layout for diagram only
    display_diagram_viewer_section()

    # Chat interface
    display_query_chat_interface()

```

- **Structured Output Display:** Uses Streamlit's markdown and st.info/st.success blocks to display **LLM-generated explanations** and structured analysis results clearly, maintaining the aesthetic defined by the Royal Blue theme.
- **Template Structure (Conceptual Streamlit 'Views'):**

Streamlit Page/View	**Corresponding Python Function**	**Description**
Main Analysis	main_analysis_page()	Core view for upload, analysis summary, and primary diagram.
Code Queries	code_queries_page()	View for diagram-aware chat interaction with the LLM.
Diagram Viewer	display_diagram_viewer_section()	Embeds diagram image and live DOT code editor.
Export Section	display_export_section()	Displays download buttons for PNG, SVG, MD, DOT formats.

SMARTBRIDGE
Let's Bridge the Gap

MILESTONE 5: Testing & Optimization

This milestone focuses on validating the **reliability, accuracy, and structural consistency** of the code-to-diagram pipeline. The system is tested using diverse code samples (Python, Java, complex/simple logic) to ensure the generated diagrams and AI summaries accurately reflect the source code. Optimization efforts center on error handling, layout stability, and LLM response quality.

Activity 5.1: Testing with Real-World Code Inputs Across Domains

Conduct end-to-end testing using varied code structures and sizes for:

- **Python Object-Oriented Code:** Test nested classes, multiple inheritance, and complex class method calls to validate the **UML Class Diagram** generation and analyzer.py's call graph mapping.
- **Java Enterprise Code:** Test interfaces, abstract classes, and method modifiers (public/private/static) to confirm diagram_generator.py correctly handles **Java UML conventions**.
- **Complex Control Flow/Functions:** Test functions with deep if/else nesting and loop structures to validate the output of the optional **Simple Diagram** logic.

Activity 5.2: Validation of Key Features and Error Handling

- **LLM Accuracy:** Evaluate llm_integration.py responses for technical accuracy, conciseness, and contextual awareness, particularly when asked about dependencies detected by analyzer.py.
- **Renderer Stability:** Introduce synthetic syntax errors into the DOT code editor to verify that regenerate_diagram_from_edited_code robustly catches **Graphviz syntax errors** and displays them cleanly in the Streamlit error box.
- **Export Integrity:** Validate that all four export formats (PNG, SVG, MD, DOT) are generated correctly and that the Markdown file accurately embeds the diagram image.

Activity 5.3: UI/UX & Styling Optimization

- **Theming Consistency:** Review the application across different viewport sizes to ensure the custom styles.css is applied consistently, and the design remains **responsive** (e.g., chat bubbles, metric cards).
- **Performance Check:** Measure the load time for parsing and analyzing very large code files (1000+ LOC) to identify bottlenecks and ensure the spinner functions in app.py correctly manage user expectation during processing.

SMARTBRIDGE
Let's Bridge the Gap

Conclusion

The **Code-to-Diagram Generator** redefines the way developers and technical writers approach code comprehension and documentation by delivering **accurate, automated, and AI-driven visualizations** of software structure and logic. By combining the power of **Streamlit** for a responsive UI, robust **Graphviz** rendering, and fast **Groq LLM interpretation**, the system intelligently processes source code (Python, Java, Jupyter) to generate context-aware artifacts like UML Class Diagrams and Flowcharts.

From the modular Python analysis pipeline (parser_module.py, analyzer.py) to the efficient generation engine (diagram_generator.py) and a user-centric interface (app.py), the system has been built with a focus on **developer productivity, maintainability, and clarity**. The Streamlit frontend ensures modular, stateful component management, while the custom-themed UI offers an intuitive and clean environment for users to upload, preview, edit, and export diagrams in real time.

Each module—from structural parsing to the interactive query chat—is designed to **simplify the process of understanding complex codebases** and enhance documentation through automation and visualization. Whether it's rapid onboarding for a new engineer or validating the logic flow in a code review, the Code-to-Diagram Generator acts as a **smart static analysis assistant**, saving time and effort while ensuring documentation remains current with the codebase.

With its seamless fusion of generative AI for semantic interpretation and specialized web technologies for visualization, the Code-to-Diagram Generator stands as a practical example of how AI can enhance core software development processes, bringing **intelligent structural insights** to developers' fingertips—fast, accurate, and ready for immediate integration.