PL/SQL to PySpark Conversion using LLMs: A Detailed Flow

1. Introduction

The migration of legacy data systems often involves transitioning from proprietary procedural languages like PL/SQL to modern, scalable frameworks such as Apache Spark, utilizing PySpark for data processing. This document outlines the detailed process of a Streamlit-based application designed to automate the conversion of PL/SQL code blocks into idiomatic PySpark DataFrame API code, powered by advanced Large Language Models (LLMs). The application aims to significantly reduce manual effort, accelerate migration timelines, and ensure the generation of clean, production-ready PySpark code.

2. Architecture and Components

The application's architecture is modular, comprising several key components that work in concert to achieve the conversion.

- **Streamlit UI:** Provides an intuitive web interface for user interaction, input, and output visualization.
- **Environment Configuration:** Securely loads API keys and other configurations for LLM providers.
- **LLM Integration (Gemini & Azure OpenAI):** Connects to and leverages the capabilities of selected LLMs for code transformation.
- **Code Chunking Module:** Intelligently breaks down large PL/SQL scripts into manageable blocks.
- **Conversion Engine:** Orchestrates the LLM calls for each chunk and stitches the results.
- **Linting Module:** Ensures the quality and adherence to Python best practices for the generated PySpark code.
- **Data Handling & Download:** Facilitates input handling (file upload/paste) and output downloads (PySpark code, chunk analysis).

3. Detailed Workflow

The conversion process follows a systematic, multi-step approach as depicted in the flow below:

3.1. Initialization and Configuration

1. **Environment Variable Loading:**
    - The application first loads critical environment variables (e.g., OPENAI_API_KEY, GEMINI_API_KEY, DEPLOYMENT_NAME, MODEL_NAME) from a .env file using python-dotenv. This ensures secure handling of sensitive credentials.

2. **LLM Client Initialization:**
    - Based on the loaded environment variables, the appropriate LLM clients are initialized:
        - **Gemini:** If GEMINI_API_KEY is present, google.generativeai.configure is called, and a GenerativeModel("gemini-1.5-pro") instance is created.
        - **Azure OpenAI:** If OPENAI_API_KEY is present, openai library's api_type, api_key, api_base, and api_version are set, making it ready for API calls.

3.2. User Interface (Streamlit)

1. **Page Configuration:** Sets up the Streamlit page title and layout for an optimal user experience.
2. **Sidebar Controls:**
    - **LLM Provider Selection:** Users can choose between "Gemini" and "Azure OpenAI" as their preferred LLM for conversion. This allows flexibility and leverages different LLM capabilities.
    - **Input Method:** Users select how to provide the PL/SQL code:
        - **Upload .sql File:** A file uploader allows users to directly upload PL/SQL scripts.
        - **Paste Code:** A text area is provided for users to paste their PL/SQL code directly.
    - **Linting Option:** A checkbox allows users to enable or disable linting of the final PySpark output, ensuring code quality.
3. **API Key Status Display:** Informative indicators in the sidebar show whether Gemini and OpenAI API keys are successfully loaded, providing transparency to the user.

3.3. PL/SQL Code Ingestion

1. **Input Acquisition:** The application reads the PL/SQL code based on the user's chosen input method (uploaded file or pasted text).
2. **Original Code Display:** The ingested PL/SQL code is immediately displayed in a dedicated section using st.code for user verification.

3.4. Intelligent Code Chunking

1. **Purpose:** To handle large PL/SQL scripts efficiently and to overcome LLM token limits, the chunk_sql_by_lines function is employed.
2. **Logic:**
    - The PL/SQL code is split into individual lines.
    - It iterates through lines, buffering them into a buffer list.
    - A max_lines parameter (defaulting to 100) acts as a soft limit for chunk size.
    - **Nesting Awareness:** A nesting counter tracks BEGIN/END blocks to avoid splitting code within logical units, ensuring semantic integrity of chunks.
    - **Delimiter-based Splitting:** Chunks are finalized when max_lines is reached, nesting is zero, and a semicolon (;) is found at the end of a line, indicating a potential end of a statement or block.

    - Any remaining lines in the buffer are added as the final chunk.
3. **Chunk Visualization & Download:**
    - The generated chunks are displayed in a DataFrame preview (st.dataframe).
    - Users can download these chunks as a CSV file (plsql_chunks.csv) for analysis or record-keeping.

3.5. LLM-Powered Conversion

1. **Iterative Conversion:** Each PL/SQL chunk is then processed independently by the selected LLM.
2. **LLM Selection:**
    - If "Gemini" is chosen, the convert_with_gemini function is called.
    - If "Azure OpenAI" is chosen, the convert_with_openai function is invoked.
3. **Prompt Engineering (Key to Quality):**
    - **Gemini Prompt:** Designed to act as a "senior data engineer," emphasizing retention of business logic, idiomatic PySpark, handling of control structures, and strict output format (only executable Python, no comments/markdown).
    - **OpenAI Prompt:** Similar, focusing on generating valid, executable Python code without explanations or markdown.
    - These prompts are crucial for guiding the LLM to produce high-quality, relevant PySpark code.
4. **Error Handling:** Both conversion functions include try-except blocks to catch potential API errors and return informative messages, preventing application crashes.
5. **Progress Indicator:** A st.spinner provides real-time feedback to the user during the conversion process, indicating which LLM is being used and the number of chunks being processed.

3.6. PySpark Code Assembly and Output

1. **Merging Converted Blocks:** The individual PySpark code snippets generated for each chunk are concatenated, separated by a clear delimiter (# ———— Next Block ————), to form the complete PySpark script.
2. **Final Code Display:** The entire generated PySpark code is displayed in a dedicated section using st.code(language="python").
3. **Download PySpark Code:** A download button (st.download_button) allows users to download the complete PySpark script as a .py file.

3.7. Code Linting (Quality Assurance)

1. **Conditional Execution:** If the "Enable Lint Final PySpark Output" checkbox is selected, the lint_code function is executed.
2. **Linting Process:**
    - The generated PySpark code is temporarily written to a file (/tmp/temp_lint.py).
    - The flake8 linter is invoked as a subprocess to analyze the code for style guide violations and programming errors.

    - The stdout of the flake8 command, which contains linting results, is captured.
3. **Linting Result Display:** The output from flake8 (or a "No lint issues found" message) is displayed, providing immediate feedback on code quality.
4. **Error Handling:** The lint_code function includes error handling for potential issues during the linting process.

4. Technical Considerations and Best Practices

- **Scalability:** Chunking large PL/SQL files enables the processing of extensive codebases without hitting LLM token limits, making the solution scalable.
- **Modularity:** The clear separation of concerns (UI, chunking, conversion, linting) makes the codebase maintainable and extensible.
- **Prompt Engineering:** The quality of the generated PySpark code is heavily dependent on well-crafted and precise prompts. Continuous refinement of these prompts is crucial.
- **Error Handling:** Robust error handling at each stage (API calls, file operations) ensures the application's stability.
- **User Experience:** Streamlit's intuitive interface, progress indicators, and clear output formatting contribute to a positive user experience.
- **Security:** Using environment variables for API keys enhances security by keeping sensitive information out of the codebase.

5. Future Enhancements

- **Interactive Editing of Chunks:** Allow users to modify individual PL/SQL chunks before sending them to the LLM.
- **PySpark Code Refinement:** Implement a post-processing step to further refine the generated PySpark code (e.g., variable renaming, common pattern optimization).
- **Cost Monitoring:** Integrate mechanisms to track API usage and costs for LLM calls.
- **Support for More LLMs:** Expand to include other LLM providers (e.g., Cohere, Anthropic).
- **Advanced Linting/Code Formatting:** Integrate more sophisticated linters or auto-formatters (e.g., Black, Pylint).
- **Dependency Management:** Suggest or automatically include necessary PySpark imports based on the converted code.

6. Conclusion

This PL/SQL to PySpark conversion application, powered by LLMs, represents a significant step towards automating legacy code migration. By intelligently chunking PL/SQL code, leveraging the code generation capabilities of LLMs, and incorporating quality assurance through linting, the solution provides a powerful and efficient tool for data engineers and organizations undergoing modernization initiatives. The modular design and user-friendly interface make it a valuable asset in accelerating the transition to modern data processing frameworks.
