Architecture of the Bank Statement Parser Agent

1 Architecture Overview

The `agent.py` file implements a **Bank Statement Parser Agent** using a modular architecture based on the **LangGraph framework**. The agent autonomously generates, tests, and refines Python parsers for bank statement PDFs. It leverages a **StateGraph** to orchestrate the process, with nodes representing distinct tasks (planning, code generation, testing, and fixing) and edges defining the flow between them. The architecture ensures modularity, robust error handling, and iterative improvement, making it both extensible and reliable.

2 Key Components

2.1 AgentState (TypedDict)

Defining the state schema that flows through the graph, `AgentState` ensures all nodes operate on consistent data. Its fields include:

- `target_bank`: Name of the bank (e.g., "icici").
- `pdf_path`: Path to the input PDF file.
- `csv_path`: Path to the reference CSV defining the expected schema.
- `parser_code`: Generated parser code.
- `error_message`: Error details from failed attempts.
- `attempts`: Current attempt count.
- `max_attempts`: Maximum allowed attempts (default: 3).
- `success`: Boolean indicating parsing success.
- `plan`: Planning output from the model.
- `debug_info`: Debugging information for failures.

2.2 StateGraph (LangGraph)

The workflow is a directed graph where nodes represent processing steps, and edges define transitions. The nodes are:

- **plan**: Generates a high-level plan for parser creation.
- **generate_code**: Produces Python parser code using the Gemini model.
- **test_code**: Tests the generated parser against the reference CSV.
- **fix_code**: Refines the parser code if tests fail.

The edges follow this flow:

- Sequential: `START` $\rightarrow$ `plan` $\rightarrow$ `generate_code` $\rightarrow$ `test_code`.
- Conditional from `test_code`:

  - If `success` is `True` or `attempts >= max_attempts`, proceed to `END`.
  - Otherwise, proceed to `fix_code`.
- Loop: `fix_code` $\rightarrow$ `generate_code` for iterative refinement.

2.3 Workflow Visualization

The workflow can be visualized as a flowchart:

- **START** leads to **Plan**, which generates a strategy.
- **Plan** connects to **Generate Code**, producing the parser.
- **Generate Code** links to **Test Code**, which validates the parser.
- From **Test Code**:
  - If successful or max attempts reached, it connects to **END**.
  - If failed and attempts remain, it loops to **Fix Code**.
- **Fix Code** loops back to **Generate Code** for refinement.

This structure ensures a clear, iterative process for parser development.

3 Node Design

3.1 Plan Node

The `plan` node uses the Gemini model to generate a text-based plan for parser creation.

- **Input**: `target_bank`, `pdf_path`, `csv_path`.
- **Output**: Updates `plan` in the state.
- **Error Handling**: Logs and returns errors if the Gemini API fails.

3.2 Generate Code Node

The `generate_code` node produces Python code for a `parse_pdf` function that extracts transaction data into a pandas DataFrame.

- Ensures the code matches the schema in `csv_path` using `pdfplumber`.
- Writes the code to `custom_parser/<target_bank>_parser.py`.
- Cleans the Gemini response to remove markdown or non-Python text.
- **Output**: Updates `parser_code` and increments `attempts`.
- **Error Handling**: Validates file creation, permissions, and non-empty code.

3.3 Test Code Node

The `test_code` node dynamically imports and executes the generated parser.

- Validates the output DataFrame against the reference CSV for schema and content equality.

- Uses `difflib` to generate detailed debug information on mismatches.
- **Output**: Updates `success`, `error_message`, and `debug_info`.
- **Error Handling**: Catches import errors, file access issues, and DataFrame mismatches.

3.4 Fix Code Node

The `fix_code` node refines the parser code using the Gemini model based on `error_message` and `debug_info`.

- **Output**: Updates `parser_code` with fixed code.
- **Error Handling**: Ensures the fixed code is non-empty and logs API failures.

4 Modularity and Clarity

- Each node is a self-contained asynchronous function, facilitating easy extension or modification.
- The `AgentState` ensures type safety and consistent data flow across nodes.
- Comprehensive logging at each step enhances transparency and aids debugging.
- The graph is compiled using `graph.compile()`, ensuring a robust execution pipeline.