Technical Documentation

Technical Documentation: Control Review Agent

1. Overview

The Control Review Agent is a LangChain-based conversational AI assistant designed for analyzing and reviewing internal bank controls. It leverages a Large Language Model (LLM) to understand user queries, interact with a control library, and perform various analytical tasks on control data. The agent is equipped with tools to filter controls, conduct reviews (5W, Operational Effectiveness, Design Effectiveness), explain its methodologies, and allow dynamic customization of its review prompts.

2. Architecture and Workflow

The agent operates through a sequence of interactions involving user input, agent processing, tool execution, and LLM-generated responses.

```
User Input ⟶ Interactive Interface (interactive_chat.py)
   |
   v
Agent Core (agent.py)
   |⟶ AgentExecutor dispatches to LLM or Tools
   |      |
   |      v
   |   LLM (ChatAnthropic via LangChain)
   |      | (Processes input, decides on tool use, generates responses)
   |      v
   |   Tools (tools.py)
   |      |⟶ FilterControls ⟶ DataLoader (data_loader.py) ⟶ controls.json
   |      |⟶ BatchReviewControls ⟶ LLMChains (for 5W, OE, DE) ⟶ Prompts
(prompts.py)
```

```
   |      |--> ExplainMethods --> LLMChain (for methods) --> Prompts (prompts.
py)
   |      |--> UpdatePromptTool --> Prompts (prompts.py) & updates LLMChains
in tools.py
   |
   v
Response --> Interactive Interface --> User
```

Workflow Steps:

1. **User Interaction:** The user interacts with the agent primarily through `src/examples/interactive_chat.py`, which provides a command-line interface. Alternatively, `src/examples/sample_run.py` executes predefined scenarios.
2. **Input to Agent:** User input is passed to the `AgentWrapper` instance in `src/agent.py`.
3. **Agent Execution:**
   - The `AgentWrapper` maintains a basic chat history and invokes the `AgentExecutor`.
   - The `AgentExecutor` in `src/agent.py` is the core orchestrator. It uses a prompt template that includes the system persona, chat history, user input, and a placeholder for agent scratchpad (intermediate tool outputs).
   - This combined prompt is sent to the LLM (`ChatAnthropic`), which has been configured and bound with available tools (`llm.bind_tools(TOOLS)`).
4. **LLM Processing & Tool Decision:**
   - The LLM processes the input and decides whether to respond directly or to use one of its bound tools.
   - If a tool is needed, the LLM generates tool invocation requests.
5. **Tool Execution (`src/tools.py`):**
   - `FilterControls`:
     - Parses user input (which can be a control ID, list of IDs, or a dictionary of attribute filters).
     - Calls `actual_filter_controls` from `src/data_loader.py`.

    - `data_loader.py` uses a Pandas DataFrame (loaded from `controls.json`) to perform case-insensitive substring matching for attribute filters or ID matching.
    - Returns a list of matching control objects.
- **`BatchReviewControls`**:
    - Takes a list of control objects and a list of review types (e.g., `["5W", "OE"]`).
    - For each control and review type, it invokes the corresponding `LLMChain` (e.g., `chain_5w`, `chain_oe`, `chain_de`).
    - These chains are defined in `src/tools.py` and use `PromptTemplate` objects from `src/prompts.py`.
    - The results of the reviews are aggregated and returned.
- **`ExplainMethods`**:
    - Invokes the `chain_methods` (an `LLMChain`) which uses a specific prompt from `src/prompts.py` to generate an explanation of the 5W, OE, and DE analysis methodologies.
- **`UpdatePromptTool`**:
    - Allows the user to dynamically change the template string for a given prompt key (e.g., "5W").
    - Calls `prompts.update_prompt()`, which updates the `PromptTemplate` object in the `PROMPT_TEMPLATES` dictionary within `src/prompts.py`.
    - Crucially, this tool also updates the `.prompt` attribute of the corresponding `LLMChain` (e.g., `chain_5w.prompt = updated_template`) stored in the `ANALYSIS_CHAINS` dictionary in `src/tools.py`. This ensures that subsequent reviews use the modified prompt.

6. **Response Generation:**
    - If a tool was used, its output (the "agent_scratchpad") is fed back into the LLM along with the original input and history.
    - The LLM then generates the final textual response to the user.

7. **Output to User:** The agent's response is printed to the command-line interface.

3. Key Components

3.1. `controls.json`

- **Purpose:** The primary data source for the agent, containing a library of internal controls.
- **Format:** A JSON file, expected to be a list of dictionaries, where each dictionary represents a control and its attributes (e.g., `control_id`, `description`, `owner`, `category`, `status`, etc.).
- **Location:** Must be present in the project root directory (`control-1/`).

3.2. `src/data_loader.py`

- **Purpose:** Responsible for loading control data from `controls.json` into a Pandas DataFrame and providing filtering capabilities.
- **Loading:**
  - Reads `controls.json` upon module initialization.
  - Stores the data in a global Pandas DataFrame (`_df_controls`) for efficient filtering.
  - Includes error handling for file not found or JSON decoding issues.
- **Filtering (`filter_controls` function):**
  - Accepts optional `control_ids` (list of strings) or `filters` (dictionary of attribute-value pairs).
  - If `control_ids` are provided, it filters the DataFrame for exact matches (case-insensitive for string IDs).
  - If `filters` are provided, it iterates through attribute-value pairs, performing case-insensitive substring searches on the respective DataFrame columns.
  - Returns a list of control dictionaries matching the criteria.

3.3. `src/prompts.py`

- **Purpose:** Defines and manages the `PromptTemplate` objects used by the LLM for various analysis tasks.
- **Structure:**
  - `INITIAL_PROMPTS`: A dictionary storing the initial string templates for "5W", "OE", "DE", and "METHODS" reviews. These templates include placeholders like `{control}` for injecting control data.
  - `PROMPT_TEMPLATES`: A dictionary where keys are prompt types (e.g., "5W") and values are `PromptTemplate` objects created from the initial string templates.
  - Individual global variables (e.g., `prompt_5w`) are also exported for convenience, though using `get_prompt()` or accessing via chains is preferred.
- **Key Functions:**
  - `get_prompt(prompt_key)`: Retrieves a `PromptTemplate` object for a given key.
  - `update_prompt(prompt_key, new_template_string)`: Updates the template string for a specified `prompt_key`. It recreates the `PromptTemplate` object in `PROMPT_TEMPLATES` and also updates the corresponding global prompt variable. This function is used by the `UpdatePromptTool`.

3.4. `src/tools.py`

- **Purpose:** Defines the custom tools available to the LangChain agent and configures the LLM client and analysis chains.
- **LLM Configuration:**
  - Retrieves `ANTHROPIC_API_KEY`, `ANTHROPIC_MODEL_NAME`, `ANTHROPIC_TEMPERATURE`, and `ANTHROPIC_MAX_TOKENS` from environment variables (with defaults).
  - Initializes the `ChatAnthropic` LLM client (`llm`).
- **Analysis Chains (`LLMChain`):**
  - Creates `LLMChain` instances for each analysis type: `chain_5w`, `chain_oe`, `chain_de`, and `chain_methods`.

    - Each chain combines the configured `llm` with its respective `PromptTemplate` from `src/prompts.py`.
    - These chains are stored in the `ANALYSIS_CHAINS` dictionary, which is used by the `UpdatePromptTool` to dynamically update the prompt used by a chain.
- **Tool Definitions:**
    - **`FilterControls` (`filter_tool`):**
        - Wraps the `filter_controls_tool_func` which intelligently parses the input string (expecting JSON for complex filters or direct string/list for IDs) and calls `actual_filter_controls` from `data_loader.py`.
        - Description clearly states how to filter by attributes or `control_id`.
    - **`BatchReviewControls` (`review_tool`):**
        - Wraps `batch_review_func`.
        - Expects a JSON string input containing a list of `controls` (control objects) and a list of `review_types`.
        - Iterates through controls (max 10) and review types, calling `single_review`.
        - `single_review` dispatches to the appropriate `LLMChain` (e.g., `chain_5w.run(control=control_data)`).
        - Aggregates and returns results.
    - **`ExplainMethods` (`methods_tool`):**
        - Wraps `explain_methods_func`.
        - Runs the `chain_methods` to provide explanations of 5W, OE, DE analyses.
    - **`UpdatePromptTool` (`@tool update_prompt_tool`):**
        - Allows runtime modification of prompt templates.
        - Takes `prompt_key` (e.g., "5W") and `new_template_string`.
        - Calls `prompts.update_prompt()` to change the template in `src/prompts.py`.
        - Crucially, it also updates the `.prompt` attribute of the corresponding `LLMChain` in the `ANALYSIS_CHAINS` dictionary (e.g., `ANALYSIS_CHAINS["5W"].prompt`

`= new_prompt_object`). This ensures the live chain uses the new prompt immediately.

- `TOOLS` **List:** Exports a list of all defined tool objects for the agent.

3.5. `src/agent.py`

- **Purpose:** Initializes and configures the LangChain agent, including the LLM, tools, prompt structure, and the agent execution logic.
- **LLM and Tool Binding:**
  - Imports `TOOLS`, `MODEL_NAME`, `TEMPERATURE`, `MAX_TOKENS` from `.tools`.
  - Initializes `ChatAnthropic` LLM.
  - Binds the `TOOLS` to the LLM using `llm.bind_tools(TOOLS)`. This makes the LLM aware of the tools and their descriptions, enabling it to decide when to use them.
- **System Persona & Prompt Template:**
  - `system_message_content`: Defines the agent's persona and capabilities. Tool descriptions are not explicitly listed here as `bind_tools` handles their availability to the LLM.
  - `prompt`: A `ChatPromptTemplate` is constructed using `MessagesPlaceholder` for `chat_history` (optional) and `agent_scratchpad` (for tool outputs), along with the system message and human input. This structure is standard for tool-calling agents.
- **Tool Calling Runnable (`tool_calling_runnable`):**
  - A LangChain Expression Language (LCEL) chain that:
    - Takes user `input`, `intermediate_steps` (tool outputs formatted by `format_to_tool_messages`), and `chat_history`.
    - Passes these to the `prompt`.
    - Pipes the formatted prompt to `llm_with_tools`.
    - The output from the LLM (which might include tool calls) is then parsed by `OpenAIToolsAgentOutputParser()`.

- **Agent Executor (`agent_executor`):**
  - An `AgentExecutor` instance is created with the `tool_calling_runnable` as the `agent` and the `TOOLS` list.
  - `verbose=True` enables logging of agent steps.
  - The `AgentExecutor` handles the loop of: LLM call → tool invocation (if any) → LLM call with tool output → final response.
- `AgentWrapper` **Class:**
  - A simple wrapper around `agent_executor` to provide a `run(input_str)` method, similar to older LangChain agent interfaces.
  - Manages a basic list-based `chat_history` (tuples of "human" and "ai" messages).
  - The `run` method invokes `self.executor.invoke()` with the input and chat history.
  - It then robustly extracts the textual output from the response dictionary.
- `agent` **Instance:** An instance of `AgentWrapper` is created and exported for use by example scripts.

3.6. `src/examples/interactive_chat.py`

- **Purpose:** Provides a command-line interface for users to interact with the agent in real-time.
- **Setup:**
  - Loads environment variables from `.env` located at the project root.
  - Imports the `agent` instance from `src.agent`.
- **Interaction Loop:**
  - Prints a welcome message and example control data.
  - Enters a `while True` loop to continuously prompt the user for input (`You:`).
  - Allows users to type "exit" or "quit" to end the session.
  - Sends the user's input to `agent.run(user_input)`.
  - Prints the agent's response.

- Includes basic error handling for `KeyboardInterrupt` and other exceptions.

4. Setup and Running

Refer to the `README.md` for detailed setup instructions, including:

- Python environment setup (virtual environment).
- Installation of dependencies from `requirements.txt`.
- Configuration of the `.env` file with `ANTHROPIC_API_KEY`.
- Commands to run `sample_run.py` and `interactive_chat.py` as modules.

5. Environment Variables

The agent uses the following environment variables, typically defined in a `.env` file in the project root:

- `ANTHROPIC_API_KEY` **(Required):** Your API key for Anthropic Claude. The agent will not function without this.
- `ANTHROPIC_MODEL_NAME` **(Optional):** Specifies the Claude model to use.
  - Default: `"claude-3-haiku-20240307"`
- `ANTHROPIC_TEMPERATURE` **(Optional):** Controls the randomness of the LLM's output. Lower values are more deterministic.
  - Default: `0.2`
- `ANTHROPIC_MAX_TOKENS` **(Optional):** The maximum number of tokens the LLM can generate in a single response.
  - Default: `4096`

6. Extensibility

- **Adding New Tools:**
  1. Define the tool function and `Tool` object in `src/tools.py`.
  2. Add the new tool object to the `TOOLS` list in `src/tools.py`.

3. Update the agent's system persona in `src/agent.py` if necessary to inform it about the new capability.

- **Adding New Prompts/Review Types:**
  1. Add the new prompt template string to `INITIAL_PROMPTS` in `src/prompts.py`.
  2. If it's a new analysis type requiring an `LLMChain`, create the chain in `src/tools.py` and add it to `ANALYSIS_CHAINS` (if you want its prompt to be updatable by `UpdatePromptTool`).
  3. Modify or add tools in `src/tools.py` to utilize this new prompt/chain.
- **Changing Control Data:**
  - Modify or replace the `controls.json` file. Ensure the new file follows the expected format (list of control dictionaries). The `data_loader.py` will automatically pick up the changes on the next run (as it loads the file at module import).

This technical documentation should provide a comprehensive understanding of the Control Review Agent's inner workings.