Documentation: AI Lab Generation Tool

Author: Kushagra Trivedi, IIIT Hyderabad
Organization: Vlabs

1. Abstract

This document provides a comprehensive overview of the "AI Lab Generation Tool" project. This tool is a web-based application designed to assist educators and students in the field of Artificial Intelligence by automating the creation of lab assignments, tutorials, and related documentation. The project aims to provide a seamless, intuitive interface for generating structured, high-quality educational content based on user-defined parameters.

2. Features

- **Dynamic Content Generation:** Automatically generates lab assignments and documentation from a set of predefined templates and user inputs.
- **Customizable Templates:** Users can create, edit, and manage templates for different types of labs and topics.
- **User-Friendly Interface:** A clean and intuitive web interface for easy interaction and content generation.
- **Markdown Support:** Full support for Markdown to format text, code blocks, and images within the generated documents.
- **Export Functionality:** Allows users to export the final documentation in various formats, such as Markdown (`.md`) or PDF.
- **Modular Codebase:** The project is structured to be easily maintainable and extensible for future features.
- **Retrieval-Augmented Generation (RAG):** Enhances content generation by retrieving relevant information from existing documents and knowledge bases.
- **Responsive Design:** Features a modern UI with dark/light mode support and responsive components.

3. Codebase Overview

The project follows a Python-based architecture with a modular design:

Main Directory Structure

- **`Pipeline 2.0/`** : Main project directory.
  - `main.py` : The entry point of the application. Initializes components and orchestrates the workflow.
  - `utils.py` : Contains utility functions used throughout the application, including text processing, file handling, and common helper functions.
  - `ui.py` : Implements the main Streamlit user interface, defining the page layout and interaction flow.
  - `ui_components.py` : Contains reusable UI components and widgets used in the main interface.
  - `BaseAgent.py` : Defines the base agent class for AI interactions, providing a common interface for various AI models and services.
  - `styles.py` : Manages UI theming, CSS components, and responsive design for a cohesive user experience.
  - `requirements.txt` : Lists all Python package dependencies required to run the project.
  - `.env` : Environment variables file for storing API keys and configuration settings (not included in repository).

RAG Components

- **`rag/`** : Directory containing Retrieval-Augmented Generation components.
  - `__init__.py` : Makes the RAG module importable and defines package-level variables and functions.
  - `rag_agent.py` : Implements the main RAG agent that combines retrieval and generation capabilities.
  - `rag_manager.py` : Manages the overall RAG workflow, coordinating between document store, retrieval, and generation.
  - `document_store.py` : Implements the storage and retrieval mechanisms for documents, including vectorization and semantic search.
  - `config.py` : Contains configuration settings for the RAG system, including model parameters and retrieval settings.
  - `rag_tester.py` : Provides testing functionality for RAG components to verify proper operation.

4. Working with the Codebase

Code Architecture

The application follows a layered architecture:

1. **Presentation Layer**: Implemented in `ui.py`, `ui_components.py`, and `styles.py`, responsible for user interaction and visual presentation.
2. **Business Logic Layer**: Implemented in `main.py`, `BaseAgent.py`, and the RAG components, responsible for processing requests and generating content.
3. **Data Layer**: Implemented in `document_store.py` and integrated with external AI services for data storage and retrieval.

Development Workflow

1. **Setting Up Development Environment**:
   - Follow the installation steps in the "Installation and Setup" section.
   - Use an IDE like VSCode or PyCharm for Python development.
   - Ensure linting and formatting tools are configured (e.g., flake8, black).
2. **Making Changes**:
   - For UI changes, focus on `ui.py` and `ui_components.py`.
   - For styling changes, modify the CSS in `styles.py`.
   - For RAG-related changes, work within the `rag/` directory.
   - For general functionality changes, update `main.py` and `utils.py`.
3. **Testing Changes**:
   - Use `rag_tester.py` to test RAG-specific functionality.
   - Run the application locally with `streamlit run ui.py` to test UI changes.
   - Verify all features continue to work after making changes.

5. UI Styling System

The application features a modern, responsive UI with comprehensive theming capabilities. The styling system is centralized in the `styles.py` file, enabling consistent visual presentation across the application.

Theme Management

The UI supports both light and dark themes, managed by the following components:

- **Theme Variables**: CSS variables for colors, shadows, and gradients are defined per theme
- **Dynamic Theming**: Theme can be switched during runtime without requiring a page reload
- **Persistence**: User theme preference is stored and remembered across sessions

Key Style Components

1. **Base Styling**:
   - Global CSS variables and foundational styles
   - Background, text, and border colors following the selected theme
   - Responsive adjustments for various screen sizes
2. **UI Components**:
   - Modern cards with hover effects and gradient accents
   - Header components with background gradients
   - Progress indicators and step visualization
   - Status badges for conveying state information
3. **Chat Interface**:
   - Styled conversation bubbles for user and AI messages
   - System message styling
   - Animation effects for new messages
   - Scrollable chat container with custom scrollbar styling
4. **Streamlit Overrides**:
   - Custom styling for Streamlit's native components
   - Consistent button styling with gradient backgrounds
   - Enhanced form inputs with proper padding and borders
   - Improved file upload components

Customizing the UI

To modify the UI appearance:

1. **Changing Theme Colors**:
   - Edit the color values in the `get_theme_variables()` function in `styles.py`
   - Colors are organized by purpose (primary, secondary, accent, etc.)
2. **Modifying Components**:
   - Each component type has its own CSS section in dedicated functions
   - Changes to card styling can be made in the `get_component_css()` function
   - Chat interface styling is located in `get_chat_css()`
3. **Adding New Components**:

- Create a new function for your component's CSS
- Add your CSS to the `get_all_styles()` function to ensure it's included

Responsive Design

The UI is designed to work across devices of various sizes:

- **Media Queries**: Adjusts layout and font sizes for smaller screens
- **Flexible Components**: UI elements adapt to available space
- **Touch-Friendly**: Controls are sized appropriately for touch interaction

6. Retrieval-Augmented Generation (RAG) System

Overview

The RAG system enhances the content generation process by retrieving relevant information from existing documents and incorporating this information into the generated output. This approach combines the benefits of retrieval-based methods with generative AI to produce more accurate and contextually relevant content.

RAG Architecture

1. **Document Processing Pipeline**:
   - Documents are processed into chunks of appropriate size
   - Chunks are vectorized (converted to numerical representations)
   - Vectors are stored in a vector database for efficient retrieval
2. **Retrieval Process**:
   - User queries are converted into vector representations
   - Vector similarity search identifies relevant document chunks
   - Most relevant chunks are retrieved and prepared for context augmentation
3. **Generation Process**:
   - Retrieved information is used to augment the context for the generative model
   - The enhanced context guides the model to produce more accurate and relevant outputs
   - The generated content maintains coherence while incorporating factual information

Key Components

1. **`rag_agent.py`**:

- Implements `RAGAgent` class that combines retrieval and generation capabilities
- Handles query processing, context augmentation, and content generation
- Manages prompt engineering to optimize results

2. **`rag_manager.py`**:
   - Orchestrates the overall RAG workflow
   - Manages document indexing, retrieval, and generation processes
   - Provides interfaces for higher-level components to utilize RAG capabilities
3. **`document_store.py`**:
   - Implements document storage and retrieval mechanisms
   - Handles vectorization of documents and queries
   - Performs similarity search to find relevant documents
4. **`config.py`**:
   - Contains configuration settings for the RAG system
   - Defines parameters for chunking, vectorization, and retrieval
   - Specifies model configurations and performance tuning options

Using the RAG System

To use the RAG system in your code:

```
from rag.rag_manager import RAGManager

# Initialize the RAG manager
rag_manager = RAGManager()

# Index documents (if not already indexed)
rag_manager.index_documents(['path/to/document1.pdf', 'path/to/document2.pdf'])

# Generate content with RAG
response = rag_manager.generate_content(
    query="Write a lab assignment about reinforcement learning",
    num_results=5  # Number of relevant documents to retrieve
)

# The response will contain content enhanced with information from relevant documents
print(response)
```

Extending the RAG System

To extend or customize the RAG system:

1. **Add New Document Sources**:
    - Implement new document processing functions in `document_store.py`
    - Add support for new file formats by extending parsing capabilities
2. **Optimize Retrieval**:
    - Tune retrieval parameters in `config.py` to improve relevance
    - Implement advanced ranking or filtering mechanisms in `rag_manager.py`
3. **Enhance Generation**:
    - Modify prompt templates in `rag_agent.py` to improve generation quality
    - Implement domain-specific post-processing for specialized content

7. Agent Architecture

The AI Lab Generation Tool uses a modular agent-based architecture where specialized AI agents handle different aspects of the content generation process. Each agent inherits from a common BaseAgent class and has a specific role in the overall pipeline.

BaseAgent Framework

The `BaseAgent` class serves as the foundation for all specialized agents and provides:

- Common initialization parameters (role, basic prompt, context)
- LLM integration and management
- Prompt enhancement capabilities
- Standard output formatting and extraction

Specialized Agents

1. RequirementsAgent

Purpose: Extracts and formulates clear requirements from PDF documents.

Key Features:

- PDF extraction and processing
- Requirement prioritization and organization
- Context awareness for domain-specific knowledge

Usage:

```
req_agent = RequirementsAgent("path/to/requirements.pdf")
req_agent.set_llm(llm)
requirements = req_agent.get_output()
```

2. HumanReviewAgentForRequirement

Purpose: Refines requirements based on human feedback.

Key Features:

- Incorporates human feedback to modify requirements
- Maintains traceability between original requirements and modifications
- Preserves requirement priorities and structure

3. ImplementationAgent

Purpose: Develops implementation plans based on approved requirements.

Key Features:

- Transforms requirements into actionable implementation steps
- Generates high-level design specifications
- Creates module outlines and coding instructions

4. CodingAgent

Purpose: Generates actual code based on implementation instructions.

Key Features:

- Produces well-structured, commented code
- Handles incremental code generation based on previous code
- Ensures code is compatible with the single-file HTML constraint

5. IntegrationAgent

Purpose: Combines code modules into a unified system.

Key Features:

- Merges previous and current code modules
- Resolves conflicts between code components
- Ensures the final system functions as a cohesive whole

6. TestingAgent

Purpose: Validates code functionality through simulated execution.

Key Features:

- Simulates code execution to identify issues
- Provides detailed test reports
- Suggests corrections for failed tests

7. WebsiteDesignAgent

Purpose: Creates a complete webpage with multiple sections/tabs by combining simulation code with content.

Key Features:

- Integrates HTML, CSS, and JavaScript into a cohesive web interface
- Organizes content into logical tabs (Aim, Theory, Objective, etc.)
- Supports enhanced styling and responsive design
- Can generate content sections from simulation code
- Handles procedure generation based on simulation logic
- Supports iterative refinement through feedback

8. DocumentationAgent

Purpose: Generates comprehensive documentation in Markdown format.

Key Features:

- Creates system architecture documentation
- Provides usage guidelines and tutorials
- Documents troubleshooting tips and maintenance instructions

9. VerifierAgent

Purpose: Validates the final system against requirements.

Key Features:

- Checks functionality against original requirements
- Verifies design and usability standards
- Provides detailed feedback on compliance or issues

Agent Interaction Flow

The agents work together in a pipeline where each agent's output becomes input for subsequent agents:

1. **RequirementsAgent** processes input documents → requirements list
2. **HumanReviewAgentForRequirement** refines requirements based on feedback
3. **ImplementationAgent** converts requirements → implementation plan
4. **CodingAgent** transforms plan → executable code modules
5. **IntegrationAgent** combines code modules → unified system
6. **TestingAgent** validates system functionality
7. **WebsiteDesignAgent** creates the final UI with all components
8. **DocumentationAgent** generates comprehensive documentation
9. **VerifierAgent** performs final validation against requirements

Extending the Agent Framework

To create a new specialized agent:

1. Inherit from the BaseAgent class:

```
from BaseAgent import BaseAgent

class MySpecializedAgent(BaseAgent):
    role = "Specialized Role"
    basic_prompt = "Detailed instructions for the agent's task"

    def __init__(self, custom_param):
        super().__init__(self.role, self.basic_prompt, context=None)
        self.custom_param = custom_param

    # Override methods as needed
```

2. Implement specialized logic in the `get_output()` method:

```
def get_output(self):
    # Agent-specific processing logic
    enhanced_prompt = self.enhance_prompt()
    result = self.llm.invoke(enhanced_prompt)
    # Additional processing
    return self._extract_text_content(result)
```

3. Add any agent-specific methods and properties required for your use case.

Agent Configuration and Tuning

Each agent can be configured through:

1. **Direct parameters** passed during initialization
2. **LLM selection** using the `set_llm()` method
3. **Prompt enhancement** via the `set_prompt_enhancer_llm()` method
4. **Custom enhancement text** using `set_custom_enhancement()`

Fine-tuning agent behavior is primarily achieved through prompt engineering and parameter adjustments specific to each agent class.

8. Instructions to Run the Project

Prerequisites

- **Python 3.11**: Ensure you have Python 3.11 installed (tested on 3.11.9). Download from python.org
- **Pip**: Python package installer (usually included with Python)
- **Streamlit**: Used to run the interactive UI
- **Git**: For version control (optional if downloading directly)
- **Access to Google API**: API key required for certain features

Installation and Setup

1. **Get the Code:**
   Option 1: Clone the repository:

   ```
   git clone https://github.com/IIIT-Hyderabad/tool-ai-lab-generation-iiith.git
   cd tool-ai-lab-generation-iiith
   ```

   Option 2: Download the code from IIIT-H Research OneDrive
2. **Create and Activate a Virtual Environment:**
   It is recommended to use a virtual environment:

```
python -m venv venv

# On Windows:
venv\Scripts\activate

# On macOS/Linux:
source venv/bin/activate
```

3. **Install Dependencies:**
   Navigate to the Pipeline 2.0 directory and install the required packages:

```
cd "Pipeline 2.0"
pip install -r requirements.txt
```

4. **Configure Environment Variables:**
   Create a `.env` file in the Pipeline 2.0 directory if it doesn't exist:

```
touch .env
```

   Open the `.env` file and add the following variables:

```
GOOGLE_API_KEY=your_google_api_key_here
```

   Replace the placeholder values with your actual API keys.
5. **RAG Setup:**
   The RAG components require additional setup:
   - Ensure your `.env` file includes any required API keys for vector databases or embedding models
   - Install additional dependencies for document processing:

```
pip install pypdf sentence-transformers faiss-cpu
```

   - Prepare a document corpus for indexing:
     a. Create a `documents/` directory in the `Pipeline 2.0` folder
     b. Add PDF, TXT, or Markdown documents to this directory
     c. These documents will be indexed by the RAG system for retrieval
   - Configure RAG settings (optional):
     a. Open `rag/config.py` to adjust parameters like chunk size, embedding dimensions, etc.
     b. Modify retrieval settings based on your specific needs and hardware capabilities

Running the Application

1. **Launch the Application:**
   From the `Pipeline 2.0` directory, run:

   ```
   streamlit run ui.py
   ```

2. **Access the User Interface:**
   The application will open a new tab in your default web browser. If it doesn't open automatically, navigate to the URL displayed in the terminal (typically http://localhost:8501).
3. **Using the Application:**
   The interface provides several interactive elements:
   - **Requirements Generation**: Upload PDF files and click "Generate Requirements" to extract requirements
   - **Human Review for Requirements**: Input your review or feedback to refine the requirements
   - **Implementation Generation**: Click "Generate Implementation" to produce code based on the requirements
   - **Iterative Code Generation and Review**: Provide feedback and refine the generated code
   - **Documentation Generation**: Generate comprehensive documentation for the code
   - **Live Code Preview**: View a live preview of the generated code at http://localhost:8000
4. **Using RAG Features:**
   - Navigate to the application settings tab to enable RAG-enhanced generation
   - Upload domain-specific documents to improve the relevance of generated content
   - Adjust RAG parameters through the UI to control retrieval depth and context size
5. **Live Code Preview:**
   - When code is generated, the application automatically starts a local HTTP server
   - The generated URL (e.g., http://localhost:8000/code.html) launches in your browser
   - If the browser doesn't open automatically, manually copy the URL from the interface

Troubleshooting

- If you encounter any issues with dependencies, try updating pip:

  ```
  pip install --upgrade pip
  ```

- For API-related errors, ensure your API keys are valid and have sufficient credits
- For RAG-specific issues:
  - Check the terminal for error messages related to document processing or embedding
  - Verify that your document corpus is properly formatted and accessible

    - Try reducing chunk size or retrieval count if you encounter memory issues
- Avoid clicking buttons repeatedly as this may cause rate-limiting with the API services
- If the application seems unresponsive, check the terminal for error messages

9. Further Developments

The AI Lab Generation Tool has several avenues for future enhancements and extensions:

RAG Validation Framework

Improving the Retrieval-Augmented Generation (RAG) system with a comprehensive validation framework would ensure more accurate and reliable content generation:

1. **Ground Truth Validation**:
    - Implement automated comparison of RAG outputs against a curated set of ground truth documents
    - Calculate precision, recall, and F1 scores for retrieved content relevance
    - Track hallucination rates and factual consistency metrics
2. **User Feedback Loop**:
    - Collect explicit feedback on RAG-generated content relevance and accuracy
    - Implement mechanisms to use this feedback for iterative improvement
    - Create a feedback dashboard for monitoring RAG performance over time
3. **Source Attribution**:
    - Develop citation mechanisms to track which sources contributed to specific parts of the generated content
    - Implement confidence scores for different segments of the generated content
    - Provide transparent source links for verification by end-users

Writerside Documentation Integration

The codebase includes templates for JetBrains Writerside, a powerful documentation engine:

1. **Using Existing Templates**:
    - Navigate to the `Pipeline 2.0/Writerside/` directory to find the templates
    - Existing structure includes configuration files (`writerside.cfg`), topic files, and XML definitions
    - Build on the starter topics: `Installation.md` , `Running-the-Pipeline.md` , and `Readme.md`
2. **Extending Documentation**:
    - Add new topics to `s.tree` to extend the documentation structure

- Use the existing format in `topics/` directory as a reference
- Maintain consistent styling defined in the configuration files

3. **Publishing Options**:
   - Utilize JetBrains Writerside to generate HTML, PDF, or other documentation formats
   - Set up CI/CD pipelines to automatically regenerate documentation on code changes
   - Consider hosting the generated documentation for online accessibility

VR Lab Development

Based on the VR example in the repository, future work could focus on extending VR lab capabilities:

1. **Expanding VR Framework**:
   - Build upon the existing A-Frame based implementation (`/vr testing/index.html`)
   - Implement more interactive VR components beyond the current text switcher and slider
   - Develop reusable VR patterns for common lab interactions (e.g., data visualization, 3D manipulatives)
2. **VR-Specific Agent**:
   - Create a specialized VR-focused agent inheriting from BaseAgent
   - Develop prompt templates specifically for generating A-Frame compatible code
   - Implement VR best practices validation in the generated content
3. **Cross-Platform Compatibility**:
   - Enhance the WebXR implementation to work across various VR headsets and browsers
   - Add responsive design elements that adapt between desktop, mobile, and VR modes
   - Implement fallback mechanisms for browsers without WebXR support
4. **Performance Optimization**:
   - Develop asset optimization techniques for VR content
   - Implement progressive loading patterns for complex VR scenes
   - Create benchmarking tools to ensure smooth performance across devices

Integration with Learning Management Systems

To increase adoption and usability:

1. **LMS Plugins**:
   - Develop integration plugins for popular LMSs like Moodle, Canvas, and Blackboard
   - Create standardized export formats compatible with SCORM and xAPI
   - Implement single sign-on capabilities for seamless user experience
2. **Analytics Integration**:
   - Add learning analytics tracking to generated labs

- Develop dashboards for instructors to monitor student progress and engagement
- Implement adaptive learning features based on student performance data

By pursuing these development paths, the AI Lab Generation Tool can evolve into a more comprehensive, robust, and widely applicable solution for educational content creation.