Mainflow AI Agent System

Release 0.0.1

Emanuele Addari, Michele Bruno, Anna Setzu, Giosue Sglavo

Sep 11, 2025

CHAPTER

ONE

INSTALLATION

1.1 Requirements

- Python 3.8+
- CrewAI
- LangChain
- UV package manager
- API access for web search and AI models

1.2 Installation Steps

1. Clone the repository
2. Install dependencies using UV: `uv install`
3. Set up environment variables
4. Configure YAML files in crew config directories

1.3 Environment Variables

Set these environment variables for API access:

```
export OPENAI_API_KEY="your-openai-key"
export SERPER_API_KEY="your-serper-key"
# Add other API keys as needed
```

1.4 Project Structure

The project uses UV for dependency management and has the following structure:

- `src/mainflow/` - Main package
- `src/mainflow/crews/` - AI agent crews
- `src/mainflow/tools/` - Custom tools
- `src/mainflow/utils/` - Utility functions

CHAPTER

TWO

QUICK START

2.1 Basic Usage

```
from src.mainflow.main import MainFlow

# Initialize and run the main flow
flow = MainFlow()
result = flow.kickoff()
```

2.2 Example Flow

1. User provides input topic and preferences
2. Input validation crew validates the request
3. Planner crew creates an execution plan
4. Web crew searches for relevant information
5. Paper crew analyzes academic papers
6. Study plan crew generates final learning plan
7. Results are saved to output files

2.3 Workflow Components

The system consists of several specialized crews:

- **Input Validation Crew** - Validates and processes user input
- **Planner Crew** - Creates execution plans
- **Web Crew** - Performs web research
- **Paper Crew** - Analyzes academic papers
- **Study Plan Crew** - Generates comprehensive study plans

2.4 Configuration

Each crew has its own configuration files:

- `agents.yaml` - Defines agent roles and capabilities

- `tasks.yaml` - Defines tasks and workflows

Make sure these files are properly configured for your use case.

CHAPTER

THREE

STREAMLIT APPLICATION FUNCTIONS

This document provides detailed documentation for all functions in the `streamlit_app.py` module, based on their NumPy-style docstrings.

3.1 Evaluation Functions

3.1.1 evaluate_search_quality

`evaluate_search_quality`(*inputs*, *outputs*)

Evaluates the relevance of retrieved documents to the user's query.

Parameters:

- **inputs** (*dict*) – Dictionary containing the original user query
- **outputs** (*dict*) – Dictionary containing the retrieved documents

Returns:

- **float** – Numeric score representing relevance (0.0 to 10.0)

This function uses a custom MLflow judge to assess document relevance using three categories:

- `highly_relevant` (10.0): Documents directly related to user query with comprehensive information
- `somewhat_relevant` (5.0): Documents generally related but may lack depth or miss aspects
- `not_relevant` (0.0): Documents are off-topic, too vague, or fail to provide useful information

The evaluation is performed using Azure GPT-4.1 model with a structured prompt template.

3.1.2 evaluate_study_plan_quality

`evaluate_study_plan_quality`(*inputs*, *outputs*)

Evaluates the quality of a study plan generated by CrewAI agents.

Parameters:

- **inputs** (*dict*) – Dictionary containing student request and context
- **outputs** (*dict*) – Dictionary containing the generated study plan

Returns:

- **float** – Numeric score representing plan quality (0.0 to 10.0)

This function assesses study plan quality using structured criteria across three tiers:

- `excellent_plan` (10.0): Comprehensive, well-structured, realistic, and perfectly tailored

- `good_plan` (7.5): Solid and helpful but may lack some detail or optimization
- `poor_plan` (2.5): Inadequate, unrealistic, or doesn't address student requirements

The evaluation considers learning objectives, time allocation, study methods, assessment checkpoints, and consideration of student constraints.

3.2 Class Methods

3.2.1 MonitoringConfig Methods

`MonitoringConfig.`**`__init__`**(*self*)

Initialize the monitoring configuration.

Sets the start time for tracking execution duration using `time.time()`.

`MonitoringConfig.`**`monitoring_crew`**(*self*, *state*, *crew_output*, *crew*, *crew_name*)

Enhanced monitoring that works alongside CrewAI autolog.

Parameters:

- **state** (*object*) – Current state of the workflow containing user information
- **crew_output** (*object*) – Output from the crew execution containing results and metadata
- **crew** (*object*) – The crew instance that was executed
- **crew_name** (*str*) – Name identifier for the crew being monitored

Notes:

This method logs various metrics to MLflow including:

- Execution time and token usage
- Cost estimates for GPT-4.1 usage
- Quality scores for specific crew types

Autolog handles detailed agent/task tracing, while this provides workflow-level metrics including token usage, costs, and execution time.

3.2.2 StreamlitFlow Methods

`StreamlitFlow.`**`__init__`**(*self*, *user_input*, *progress_callback=None*, *status_callback=None*)

Initialize the StreamlitFlow with user input and optional callbacks.

Parameters:

- **user_input** (*str*) – The user's input describing their background and learning goals
- **progress_callback** (*callable, optional*) – Function to call for updating progress bar (default: None)
- **status_callback** (*callable, optional*) – Function to call for updating status messages (default: None)

`StreamlitFlow.`**`insert_topic`**(*self*)

Initialize the workflow with user input validation.

This is the starting point of the CrewAI flow that validates and sets the user's question in the workflow state.

Raises:

- **ValueError** – If the user input is invalid (contains escape sequences or is empty)

`StreamlitFlow.`**`sanitize_input`**`(`*self*`)`

Sanitize and validate user input using InputValidationCrew.

This step processes the raw user input to extract structured information about the user's background and learning goals. Updates progress to 20% and logs metrics to MLflow.

`StreamlitFlow.`**`generate_plan`**`(`*self*`)`

Generate a structured study plan outline using PlanningCrew.

Creates a comprehensive learning plan based on the sanitized user information. Updates progress to 35% and evaluates plan quality.

`StreamlitFlow.`**`web_search`**`(`*self*`)`

Search for web resources using WebCrew.

Finds relevant online learning resources, tutorials, and materials based on the generated study plan. Updates progress to 50% and evaluates search result relevance.

`StreamlitFlow.`**`paper_research`**`(`*self*`)`

Research academic papers using PaperCrew.

Searches for relevant academic papers and research materials that align with the study plan topics. Updates progress to 65%.

`StreamlitFlow.`**`define_calendar`**`(`*self*`)`

Create a study calendar using CalendarCrew.

Generates a time-based schedule that integrates web resources, academic papers, and the study plan into a practical calendar. Updates progress to 80%.

`StreamlitFlow.`**`create_study_plan`**`(`*self*`)`

Create the final comprehensive study plan using FinalStudyPlanCrew.

Combines all previous outputs (resources, papers, plan, calendar) into a cohesive, personalized study plan. Updates progress to 95% and performs final quality evaluation.

3.3 Main Application Function

3.3.1 main

`main``()`

Main Streamlit application entry point.

This function sets up and runs the complete EY Junior Accelerator Streamlit application, including page configuration, disclaimer handling, user interface components, and the AI-powered study plan generation workflow.

Application Flow:

1. Page configuration and disclaimer acceptance
2. Custom styling application
3. User interface layout (header, form, etc.)
4. Study plan generation workflow execution
5. Results display with download options
6. Error handling and user feedback

Notes:

This function manages the complete user experience from input collection to final study plan delivery, with integrated MLflow monitoring and progress tracking throughout the workflow.

3.4 Nested Functions

3.4.1 show_disclaimer

`show_disclaimer()`

Display the AI system disclaimer dialog.

Shows important information about the AI-powered nature of the system and requires user acknowledgment before proceeding with the application. Provides options to continue or cancel the session.

This function is nested within the `main()` function and uses Streamlit's dialog decorator.

3.4.2 update_progress

`update_progress`(*progress_value*)

Update the progress bar with the given value.

Parameters:

- **progress_value** (*float*) – Progress value between 0.0 and 1.0

This function is nested within the `main()` function and provides real-time progress updates during workflow execution.

3.4.3 update_status

`update_status`(*status_message*)

Update the status text display.

Parameters:

- **status_message** (*str*) – Status message to display to the user

This function is nested within the `main()` function and provides real-time status updates during workflow execution.

3.5 Workflow Progress Mapping

The StreamlitFlow methods follow a specific progress mapping:

- **0-20%**: Input sanitization and validation
- **20-35%**: Study plan outline generation
- **35-50%**: Web resource discovery
- **50-65%**: Academic paper research
- **65-80%**: Calendar creation
- **80-95%**: Final study plan assembly
- **95-100%**: Quality evaluation and completion

Each step includes comprehensive logging to MLflow for monitoring token usage, execution time, and quality metrics.

CHAPTER

FOUR

CREWS MODULES

This document provides comprehensive documentation for all crew modules in the EY Junior Accelerator system. Each crew specializes in specific aspects of the study plan generation workflow.

4.1 Input Validation Crew

The Input Validation Crew handles the validation, sanitization, and analysis of user input for academic research purposes. It includes multiple agents that work together to ensure input quality, ethics compliance, and security validation.

4.1.1 InputValidationCrew Class

`class InputValidationCrew`

Input Validation Crew for Academic Research.

This crew handles the validation, sanitization, and analysis of user input for academic research purposes. It includes multiple agents that work together to ensure input quality, ethics compliance, and security validation.

Attributes:

- **agents** (*List[BaseAgent]*) – List of agents in the crew
- **tasks** (*List[Task]*) – List of tasks to be executed by the crew
- **model** (*str*) – Azure OpenAI model configuration string

Constructor:

`__init__()`

Initialize the InputValidationCrew.

Sets up Azure OpenAI configuration and model string from environment variables.

Notes:

Requires the following environment variables:

- `AZURE_OPENAI_API_KEY` – Azure OpenAI API key
- `AZURE_OPENAI_ENDPOINT` – Azure OpenAI endpoint URL
- `AZURE_DEPLOYMENT_NAME` – Azure deployment name

Agent Methods:

`input_sanitizer()` → Agent

Create an input sanitization agent.

Returns:

- **Agent** – An agent configured to sanitize and clean user input

`ethics_checker()` → Agent

Create an ethics checking agent.

The ethics checker agent validates that the cleaned input is appropriate and complies with academic ethics standards.

Returns:

- **Agent** – An agent configured to check academic ethics compliance

`role_identifier()` → Agent

Create a role identification agent.

Returns:

- **Agent** – An agent configured to identify user roles and responsibilities

`knowledge_identifier()` → Agent

Create a knowledge domain identification agent.

Returns:

- **Agent** – An agent configured to identify relevant knowledge domains

`goals_identifier()` → Agent

Create a goals identification agent.

Returns:

- **Agent** – An agent configured to identify and analyze user goals

`security_validator()` → Agent

Create a security validation agent.

Returns:

- **Agent** – An agent configured to validate security aspects of user input

`final_validator()` → Agent

Create a final validation agent.

Returns:

- **Agent** – An agent configured to perform final validation of processed input

Task Methods:

`sanitize_input()` → Task

Create an input sanitization task.

Returns:

- **Task** – A task configured to sanitize user input

`check_academic_ethics()` → Task

Create an academic ethics checking task.

Returns:

- **Task** – A task configured to validate academic ethics compliance `validate_security()` → Task

Create a security validation task.

Returns:

- **Task** – A task configured to validate security aspects

`individuate_role()` → Task

Create a role identification task.

Returns:

- **Task** – A task configured to identify user roles

`individuate_knowledge()` → Task

Create a knowledge domain identification task.

Returns:

- **Task** – A task configured to identify knowledge domains

`final_validation()` → Task

Create a final validation task.

Returns:

- **Task** – A task configured to perform final validation

`crew()` → Crew

Create the Input Validation Crew.

Assembles all agents and tasks into a sequential crew for input validation.

Returns:

- **Crew** – A configured crew with all agents and tasks for input validation

Notes:

The crew uses sequential processing where tasks are executed one after another. Verbose mode is enabled for detailed logging.

4.2 Planner Crew

The Planning Crew specializes in creating comprehensive study plans based on user requirements and learning objectives. The crew is composed of three specialized agents that work together to define, write, and review study plans to ensure quality and effectiveness.

4.2.1 PlanningCrew Class

`class PlanningCrew`

Study Planning Crew for Academic Research and Learning.

This crew specializes in creating comprehensive study plans based on user requirements and learning objectives. The crew is composed of three specialized agents that work together to define, write, and review study plans to ensure quality and effectiveness.

Attributes:

- **agents** (*List[BaseAgent]*) – List of specialized agent instances for the crew
- **tasks** (*List[Task]*) – List of task instances to be executed by the crew

- **agents_config** (*str*) – Path to the agents configuration YAML file (config/agents.yaml)
- **tasks_config** (*str*) – Path to the tasks configuration YAML file (config/tasks.yaml)
- **model** (*str*) – Azure OpenAI model configuration string

Planning Process:

- Analyzing user requirements and learning objectives
- Defining a structured study plan framework
- Writing detailed study plan content
- Reviewing and refining the final plan

Constructor:

`__init__``()`

Initialize the PlanningCrew.

Sets up Azure OpenAI configuration and model string from environment variables. Configures the necessary API credentials and version for Azure OpenAI integration.

Raises:

- **KeyError** – If required environment variables are not set
- **ValueError** – If environment variables contain invalid values

Notes:

Requires the following environment variables:

- `AZURE_OPENAI_API_KEY` – The API key for Azure OpenAI service
- `AZURE_OPENAI_ENDPOINT` – The endpoint URL for Azure OpenAI service
- `AZURE_DEPLOYMENT_NAME` – The name of the Azure OpenAI deployment

Agent Methods:

`plan_definer``()` → Agent

Create the plan definer agent.

This agent analyzes user requirements and learning objectives to define the structure and framework for an effective study plan. It identifies key topics, prerequisites, and learning pathways.

Returns:

- **Agent** – The plan definer agent instance configured with Azure OpenAI model

Notes:

The plan definer agent is responsible for:

- Analyzing user input and requirements
- Identifying learning objectives and goals
- Defining the overall structure of the study plan
- Setting milestones and timelines
- Determining prerequisites and dependencies

`plan_writer()` → Agent

Create the plan writer agent.

This agent takes the defined plan structure and creates detailed, comprehensive study plan content with specific learning activities, resources, and step-by-step instructions.

Returns:

- **Agent** – The plan writer agent instance configured with Azure OpenAI model

Notes:

The plan writer agent is responsible for:

- Creating detailed study plan content
- Specifying learning activities and exercises
- Recommending resources and materials
- Organizing content in a logical sequence
- Adding time estimates and difficulty levels

`plan_reviewer()` → Agent

Create the plan reviewer agent.

This agent reviews the written study plan to ensure quality, completeness, and effectiveness. It validates the plan against best practices and provides feedback for improvement.

Returns:

- **Agent** – The plan reviewer agent instance configured with Azure OpenAI model

Notes:

The plan reviewer agent is responsible for:

- Reviewing plan content for quality and completeness
- Checking alignment with learning objectives
- Validating the logical flow and structure
- Ensuring appropriate difficulty progression
- Providing recommendations for improvement
- Verifying resource accessibility and relevance

Task Methods:

`define_plan()` → Task

Create the define plan task.

This task handles the analysis of user requirements and the definition of the study plan structure and framework. It establishes the foundation for the detailed plan creation.

Returns:

- **Task** – The define plan task instance using configuration from tasks_config[“define_plan”]

Notes:

This is typically the first task in the planning pipeline. It establishes the foundation for subsequent plan writing and reviewing. The task output includes the plan structure, objectives, and timeline.

`crew()` → Crew

Create the Planning Crew.

Returns:

- **Crew** – A configured crew with all agents and tasks for study plan creation

4.3 Web Crew

The Web Research Crew specializes in enhancing study plans with relevant web resources by leveraging advanced search capabilities. It takes study plans from the planning crew and systematically searches for high-quality educational resources, official documentation, tutorials, and learning materials.

4.3.1 WebCrew Class

`class WebCrew`

Web Research Crew for Educational Resource Discovery.

This crew specializes in enhancing study plans with relevant web resources by leveraging advanced search capabilities. It takes study plans from the planning crew and systematically searches for high-quality educational resources, official documentation, tutorials, and learning materials.

Attributes:

- **agents** (*List[BaseAgent]*) – List of specialized agent instances for web research
- **tasks** (*List[Task]*) – List of task instances to be executed by the crew
- **agents_config** (*str*) – Path to the agents configuration YAML file
- **tasks_config** (*str*) – Path to the tasks configuration YAML file
- **model** (*str*) – Azure OpenAI model configuration string
- **search_tool** (*SerperSearchTool*) – Custom search tool instance for web research

Web Research Process:

- Analyzing study plan topics and learning objectives
- Conducting targeted web searches using Serper search tool
- Evaluating and filtering search results for educational value
- Curating trustworthy and relevant learning resources
- Organizing resources by topic and difficulty level

Constructor:

`__init__()` → None

Initialize the WebCrew.

Sets up Azure OpenAI configuration, search tools, and environment variables required for web research operations.

Raises:

- **KeyError** – If required environment variables are not set
- **ValueError** – If environment variables contain invalid values
- **ConnectionError** – If unable to connect to Azure OpenAI or Serper API

Notes:

Requires the following environment variables:

- `AZURE_OPENAI_API_KEY` – The API key for Azure OpenAI service
- `AZURE_OPENAI_ENDPOINT` – The endpoint URL for Azure OpenAI service
- `AZURE_DEPLOYMENT_NAME` – The name of the Azure OpenAI deployment
- `SERPER_API_KEY` – The API key for Serper search service

Agent Methods:

`web_researcher()` → Agent

Create the web researcher agent.

This agent specializes in conducting intelligent web searches to find relevant educational resources based on study plan requirements. It uses the Serper search tool to perform targeted searches and evaluate results.

Returns:

- **Agent** – The web researcher agent instance configured with Serper search tool

Notes:

The web researcher agent is responsible for:

- Analyzing study plan topics to identify search queries
- Conducting systematic web searches using Serper API
- Evaluating search results for educational relevance and quality
- Filtering out inappropriate or low-quality content
- Prioritizing official documentation and trusted educational sources
- Organizing findings by topic and difficulty level

Task Methods:

`web_search_task()` → Task

Create the web search task.

This task handles the systematic search and curation of web resources based on the study plan provided by the planning crew. It focuses on finding high-quality educational materials and learning resources.

Returns:

- **Task** – The web search task instance using configuration from tasks_config[“web_search_task”]

Notes:

The web search task performs the following operations:

- Parses the input study plan to extract key topics and concepts
- Generates targeted search queries for each topic
- Executes web searches using the Serper search tool
- Evaluates and filters search results for educational value
- Curates a list of high-quality learning resources
- Organizes resources by relevance and learning progression
- Provides descriptions and recommendations for each resource

`crew()` → Crew

Create the Web Research Crew.

Returns:

- **Crew** – A configured crew with web research agents and tasks

4.4 Paper Crew

The Academic Paper Research Crew specializes in scientific literature research, focusing on identifying, extracting, and searching for relevant academic papers from ArXiv. It combines multiple agents to provide comprehensive paper discovery capabilities.

4.4.1 PaperCrew Class

`class PaperCrew`

Academic Paper Research Crew for ArXiv Paper Discovery.

This crew specializes in scientific literature research, focusing on identifying, extracting, and searching for relevant academic papers from ArXiv. It combines multiple agents to provide comprehensive paper discovery capabilities.

Attributes:

- **agents** (*List[BaseAgent]*) – List of specialized agents for paper research tasks
- **tasks** (*List[Task]*) – List of tasks to be executed by the crew

Paper Research Process:

- Identifying scientific topics from user input
- Extracting relevant keywords for academic search
- Validating search queries for optimal results
- Performing ArXiv searches and retrieving papers

Agent Methods:

`scientific_topic_identifier()` → Agent

Create a scientific topic identification agent.

This agent analyzes user input to identify the core scientific topics and research areas that should be explored.

Returns:

- **Agent** – An agent configured to identify scientific topics from text input

Notes:

The agent uses configuration from the YAML file to define its behavior and capabilities for topic identification.

`keyword_extractor()` → Agent

Create a keyword extraction agent.

This agent extracts relevant keywords and search terms from identified scientific topics to optimize ArXiv search queries.

Returns:

- **Agent** – An agent configured to extract keywords for academic search

Notes:

The agent focuses on extracting domain-specific terminology and research-relevant keywords that will improve search effectiveness.

`query_validator()` → Agent

Create a search query validation agent.

This agent validates and optimizes search queries to ensure they are well-formed and likely to return relevant results from ArXiv.

Returns:

- **Agent** – An agent configured to validate and optimize search queries

Notes:

The agent checks query syntax, relevance, and potential effectiveness before the actual search is performed.

`arxiv_searcher()` → Agent

Create an ArXiv search agent.

This agent performs the actual search on ArXiv using validated queries and returns relevant academic papers.

Returns:

- **Agent** – An agent configured to search ArXiv with arxiv_searcher_tool

Notes:

The agent is equipped with the arxiv_searcher_tool to interact with the ArXiv API and retrieve academic papers.

Task Methods:

`scientific_topic_extraction_task()` → Task

Create a scientific topic extraction task.

This task processes user input to identify and extract the main scientific topics and research areas of interest.

Returns:

- **Task** – A task configured to extract scientific topics from user input

Notes:

This is typically the first task in the pipeline, providing the foundation for subsequent keyword extraction and search operations.

`keyword_extraction_task()` → Task

Create a keyword extraction task.

This task takes identified scientific topics and extracts relevant keywords and search terms for ArXiv queries.

Returns:

- **Task** – A task configured to extract keywords from scientific topics

Notes:

The task builds upon the output of the topic extraction task to create search-optimized keywords.

`validate_query_task()` → Task

Create a query validation task.

This task validates and optimizes the extracted keywords into well-formed search queries suitable for ArXiv.

Returns:

- **Task** – A task configured to validate and optimize search queries

Notes:

The task ensures that search queries are syntactically correct and likely to return relevant academic papers.

`perform_search_task()` → Task

Create an ArXiv search task.

This task executes the validated search queries against ArXiv and retrieves relevant academic papers.

Returns:

- **Task** – A task configured to perform ArXiv searches and retrieve papers

Notes:

This is the final task in the pipeline that produces the actual search results from ArXiv.

`crew()` → Crew

Create the Paper Research Crew.

Returns:

- **Crew** – A configured crew with all agents and tasks for paper research

4.5 Study Plan Crew

The Final Study Plan Crew specializes in creating, formatting, and reviewing final study plans for academic purposes. It combines multiple specialized agents to produce comprehensive, well-structured, and visually appealing study plans that integrate all previously gathered information and resources.

4.5.1 FinalStudyPlanCrew Class

`class FinalStudyPlanCrew`

Final Study Plan Crew for Academic Learning Management.

This crew specializes in creating, formatting, and reviewing final study plans for academic purposes. It combines multiple specialized agents to produce comprehensive, well-structured, and visually appealing study plans that integrate all previously gathered information and resources.

Attributes:

- **agents** (*List[BaseAgent]*) – List of specialized agent instances for the crew
- **tasks** (*List[Task]*) – List of task instances to be executed by the crew
- **agents_config** (*str*) – Path to the agents configuration YAML file
- **tasks_config** (*str*) – Path to the tasks configuration YAML file

Final Study Plan Creation Process:

- Consolidating information from previous crews (planning, web search, papers)
- Filling in detailed study plan content with specific activities

- Creating ASCII art and visual formatting for enhanced presentation
- Reviewing and validating the final study plan for completeness and quality

Agent Methods:

`final_plan_filler()` → Agent

Create the final plan filler agent.

This agent is responsible for consolidating all information from previous crews (planning, web resources, academic papers, calendar) and filling the final study plan with comprehensive, detailed content.

Returns:

- **Agent** – The plan filler agent instance configured with settings from agents_config[“final_plan_filler”]

Notes:

The final plan filler agent performs the following functions:

- Integrates information from all previous crew outputs
- Creates detailed learning activities and milestones
- Organizes content in a logical, progressive structure
- Ensures completeness and coherence of the study plan
- Adds specific timelines and learning objectives

`ascii_writer()` → Agent

Create the ASCII writer agent.

This agent specializes in creating ASCII art, visual formatting, and enhancing the presentation of the study plan to make it more engaging and visually appealing.

Returns:

- **Agent** – The ASCII writer agent instance configured with settings from agents_config[“ascii_writer”]

Notes:

The ASCII writer agent is responsible for:

- Creating ASCII art headers and dividers
- Adding visual elements and formatting
- Enhancing readability with structured layouts
- Creating progress bars and visual timelines
- Adding decorative elements to improve engagement

`final_plan_reviewer()` → Agent

Create the final plan reviewer agent.

This agent performs comprehensive quality assurance on the final study plan, ensuring completeness, accuracy, and adherence to best practices in educational planning.

Returns:

- **Agent** – The plan reviewer agent instance configured with settings from agents_config[“final_plan_reviewer”]

Notes:

The final plan reviewer agent validates:

- Completeness of all study plan sections
- Logical flow and progression of learning activities
- Accuracy and relevance of included resources
- Appropriate time allocation and realistic timelines
- Alignment with original learning objectives
- Quality of formatting and presentation

Task Methods:

`fill_final_plan()` → Task

Create the fill final plan task.

This task handles the consolidation and integration of all information from previous crews to create a comprehensive, detailed final study plan.

Returns:

- **Task** – The fill plan task instance using configuration from tasks_config[“fill_final_plan”]

Notes:

This task is the core content creation step that:

- Combines outputs from planning, web search, papers, and calendar crews
- Creates detailed learning activities and assignments
- Establishes clear learning objectives and outcomes
- Organizes content into logical modules or sections
- Provides specific timelines and milestones

`write_ascii()` → Task

Create the write ASCII task.

This task handles the visual formatting and enhancement of the study plan with ASCII art, decorative elements, and improved layout structure.

Returns:

- **Task** – The write ASCII task instance using configuration from tasks_config[“write_ascii”]

Notes:

This task enhances the study plan with:

- ASCII art headers and section dividers
- Visual progress indicators and timelines
- Formatted tables and lists
- Decorative elements for improved engagement
- Structured layout for better readability

`crew()` → Crew

Create the Final Study Plan Crew.

Returns:

- **Crew** – A configured crew with all agents and tasks for final study plan creation

4.6 Calendar Crew

The Calendar Crew specializes in creating detailed study calendars and schedules based on study plans and learning objectives. It transforms abstract study plans into concrete, time-bound schedules with specific milestones, deadlines, and learning activities organized across time periods.

4.6.1 CalendarCrew Class

`class CalendarCrew`

Calendar Crew for Study Schedule Management.

This crew specializes in creating detailed study calendars and schedules based on study plans and learning objectives. It transforms abstract study plans into concrete, time-bound schedules with specific milestones, deadlines, and learning activities organized across time periods.

Attributes:

- **agents** (*List[BaseAgent]*) – List of specialized agent instances for calendar management
- **tasks** (*List[Task]*) – List of task instances to be executed by the crew
- **agents_config** (*str*) – Path to the agents configuration YAML file
- **tasks_config** (*str*) – Path to the tasks configuration YAML file
- **model** (*str*) – Azure OpenAI model configuration string

Calendar Creation Process:

- Analyzing study plans to identify key topics and time requirements
- Defining calendar structure with appropriate time blocks and milestones
- Writing detailed calendar entries with specific learning activities
- Organizing schedule to optimize learning progression and retention

Constructor:

`__init__()`

Initialize the CalendarCrew.

Sets up Azure OpenAI configuration and model string from environment variables. Configures the necessary API credentials and version for Azure OpenAI integration used by calendar planning agents.

Raises:

- **KeyError** – If required environment variables are not set
- **ValueError** – If environment variables contain invalid values

Notes:

Requires the following environment variables:

- `AZURE_OPENAI_API_KEY` – The API key for Azure OpenAI service
- `AZURE_OPENAI_ENDPOINT` – The endpoint URL for Azure OpenAI service
- `AZURE_DEPLOYMENT_NAME` – The name of the Azure OpenAI deployment

Agent Methods:

`calendar_definer()` → Agent

Create the calendar definer agent.

This agent analyzes study plans and defines the overall structure and framework for study calendars. It determines appropriate time blocks, learning phases, milestones, and scheduling constraints based on the study plan requirements and user preferences.

Returns:

- **Agent** – The calendar definer agent instance configured with Azure OpenAI model

Notes:

The calendar definer agent is responsible for:

- Analyzing study plan content and time requirements
- Defining calendar structure and time allocation strategies
- Identifying key milestones and checkpoint dates
- Establishing learning phases and progression stages
- Considering optimal spacing for different types of learning activities
- Setting realistic timeframes based on content complexity

`calendar_writer()` → Agent

Create the calendar writer agent.

This agent takes the defined calendar structure and creates detailed, specific calendar entries with learning activities, assignments, and milestones. It transforms the abstract calendar framework into concrete, actionable daily and weekly schedules.

Returns:

- **Agent** – The calendar writer agent instance configured with Azure OpenAI model

Notes:

The calendar writer agent is responsible for:

- Creating detailed daily and weekly schedule entries
- Specifying concrete learning activities and tasks
- Adding specific deadlines and milestone dates
- Organizing activities for optimal learning progression
- Including review sessions and spaced repetition schedules
- Balancing study load across different time periods
- Adding buffer time for review and consolidation

Task Methods:

`define_calendar()` → Task

Create the define calendar task.

This task handles the analysis of study plan requirements and the definition of the calendar structure, time allocation framework, and scheduling constraints for the study calendar.

Returns:

- **Task** – The define calendar task instance using configuration from tasks_config[“define_calendar”]

Notes:

This task is typically the first in the calendar creation pipeline. It establishes the foundation for detailed calendar content creation by defining:

- Overall time structure and learning phases
- Milestone dates and checkpoint schedules
- Time allocation strategies for different topics
- Scheduling constraints and preferences

`crew()` → Crew

Create the Calendar Crew.

Returns:

- **Crew** – A configured crew with all agents and tasks for calendar creation

4.7 Crew Workflow Integration

The crews work together in a sequential pipeline:

1. **Input Validation Crew** – Validates and sanitizes user input
2. **Planning Crew** – Creates the initial study plan structure
3. **Web Crew** – Finds relevant online resources
4. **Paper Crew** – Discovers academic papers from ArXiv
5. **Calendar Crew** – Creates time-based schedules
6. **Study Plan Crew** – Assembles the final comprehensive study plan

Each crew builds upon the output of the previous crews to create a comprehensive, personalized learning experience.

CHAPTER

FIVE

TOOLS MODULE

5.1 Custom Tool

class `src.mainflow.tools.custom_tool.`**`SerperSearchTool`**(*api_key=None*)

Bases: `BaseTool`

Parameters

`api_key` (*`str`*)

`name: str`

The unique name of the tool that clearly communicates its purpose.

`description: str`

Used to tell the model how/when/why to use the tool.

`__init__`(*api_key=None*)

Create a new model by parsing and validating input data from keyword arguments.

Raises [*ValidationError*][pydantic_core.ValidationError] if the input data cannot be validated to form a valid model.

self is explicitly positional-only to allow *self* as a field name.

Parameters

`api_key` (*`str`*)

`api_key: str`

`model_config: ClassVar[ConfigDict] = {'arbitrary_types_allowed': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

5.2 ArXiv Searcher Tool

The ArXiv Searcher Tool provides functionality to search and retrieve academic papers from the arXiv database. This tool is specifically designed for academic research and paper discovery within the EY Junior Accelerator system.

`src.mainflow.tools.custom_tool.`**`arxiv_searcher_tool`**(*arxiv_queries*)

ArXiv Searcher Tool - Performs a search on arXiv for a list of queries and returns links to the found papers.

This tool interfaces with the arXiv API to search for academic papers based on provided query strings. It returns structured information about the most relevant papers found for each query.

Parameters:

- **arxiv_queries** (*List[str]*) – A list of search strings to query the arXiv database

Returns:

- **Dict[str, List[Dict[str, Any]]]** – A dictionary where each key is a query string and the value is a list of papers found. Each paper is represented as a dictionary with the following fields:
  - `title` (*str*) – The title of the paper
  - `authors` (*str*) – Comma-separated list of author names
  - `abstract` (*str*) – The paper's abstract with newlines removed
  - `year` (*int*) – The publication year
  - `link` (*str*) – The arXiv entry ID/URL for the paper

Search Configuration:

- **Maximum Results:** 1 paper per query (most relevant)
- **Sort Criteria:** Relevance-based sorting
- **Error Handling:** Graceful handling with empty lists for failed queries

Implementation Details:

The tool uses the `arxiv` Python library to interact with the arXiv API. For each query in the input list:

1. Creates an `arxiv.Search` object with the query string
2. Configures search parameters (max_results=1, sort_by=Relevance)
3. Retrieves results using an `arxiv.Client()` instance
4. Extracts and formats paper information into a structured dictionary
5. Handles exceptions gracefully by returning empty results for failed queries

Error Handling:

If an error occurs during the search of a specific query, that query will have an empty list in the results dictionary. Error messages are printed to the console for debugging purposes.

Usage Example:

```
queries = ["machine learning", "natural language processing"]
results = arxiv_searcher_tool(queries)

# Results structure:
# {
#     "machine learning": [
#         {
#             "title": "Paper Title",
#             "authors": "Author1, Author2",
#             "abstract": "Paper abstract...",
#             "year": 2023,
#             "link": "http://arxiv.org/abs/2301.xxxxx"
#         }
#     ],
#     "natural language processing": [...]
# }
```

Notes:

- The tool is designed for integration with CrewAI agents and tasks

- Currently limited to 1 result per query to maintain performance
- Abstract text is cleaned by removing newline characters
- The tool requires an active internet connection to access arXiv
- No API key is required as arXiv provides free access to their database

Dependencies:

- `arxiv` – Python library for arXiv API access
- `typing` – For type hints and annotations
- `crewai.tools` – For CrewAI tool integration

CHAPTER

SIX

UTILS MODULE

6.1 Input Validation

`src.mainflow.utils.input_validation.`**`is_valid_input`**(*user_input*)

> **Return type**
> `bool`
>
> **Parameters**
> **`user_input`** (*`str`*)

CHAPTER

SEVEN

CONFIGURATION

7.1 Project Configuration

The project uses several configuration files:

- `pyproject.toml` - Project metadata and dependencies
- `uv.lock` - Locked dependency versions

7.2 Crew Configuration

Each crew has its own configuration directory with:

- `agents.yaml` - Agent definitions
- `tasks.yaml` - Task definitions

7.3 Agent Configuration

Agents are defined with:

- Role and goal
- Backstory and context
- Tools and capabilities
- LLM configuration

7.4 Task Configuration

Tasks define:

- Description and expected output
- Agent assignments
- Dependencies and context
- Output formats

7.5 Environment Variables

Required environment variables:

```
# OpenAI API
OPENAI_API_KEY=your_key_here

# Search API
SERPER_API_KEY=your_key_here

# Other APIs as needed
```

CHAPTER

EIGHT

EXAMPLES

8.1 Basic Study Plan Generation

```python
from src.mainflow.main import MainFlow

# Create a study plan for machine learning
flow = MainFlow()
result = flow.kickoff()
```

8.2 Advanced Usage

8.2.1 Customizing Crews

You can customize crew behavior by modifying the YAML configuration files:

```yaml
# Example agent configuration
researcher:
  role: "Research Specialist"
  goal: "Conduct thorough research on specified topics"
  backstory: "Expert researcher with deep knowledge"
```

8.2.2 Working with Results

```python
# Access generated study plan
with open('output/final_study_plan.md', 'r') as f:
    study_plan = f.read()

# Process the results
print(study_plan)
```

8.3 API Examples

8.3.1 Using Individual Crews

```python
from src.mainflow.crews.input_crew.input_validation_crew import InputValidationCrew

# Use input validation crew directly
```

(continues on next page)

(continued from previous page)

```python
crew = InputValidationCrew()
result = crew.crew().kickoff(inputs={"topic": "machine learning"})
```

8.3.2 Custom Tools

```python
from src.mainflow.tools.custom_tool import CustomTool

# Use custom tools
tool = CustomTool()
result = tool.run("search query")
```

CHAPTER

NINE

EY JUNIOR ACCELLERATOR DOCUMENTATION

Application Owner: Emanuele Addari, Michele Bruno, Anna Setzu, Giosuè Sglavo
Document Version: 1.0.0
Reviewers: /
Last Updated: 08/09/2025

9.1 Relevant Links

- **`GitHub Repository <https://github.com/eaddari/AiAcademy.git>`__**
- **Cloud Infrastructure**: Azure & Azure AI Foundry

9.2 General Information

Purpose and Intended Use: EY Junior Accelerator is a multi-agent artificial intelligence system developed to generate personalized study plans for new company hires. The system's primary objective is to analyze user profiles (role, competency level, learning objectives) and produce structured learning paths with web resources, scientific references, and temporal scheduling through the orchestration of various specialized agents.

- **Sector**: Corporate training and skills development
- **Problem Solved**: Optimizes new hire onboarding by providing targeted learning paths, reduces integration time, and improves learning effectiveness
- **Target Users**: HR managers, training managers, new employees, team leaders
- **KPIs**: Onboarding completion time, training satisfaction, new hire retention rate, learning path effectiveness
- **Ethical/Regulatory**: GDPR compliance for personal data on training profiles, EU AI Act adherence for decision-support systems in human resource management
- **Prohibited Uses**: Individual performance evaluation, discriminatory profiling based on protected characteristics, comparative ranking between employees
- **Operational Environment**: Cloud-based deployment with web interface for user interaction

9.3 Risk Classification

- **Classification**: **Limited Risk**
- **Reasoning**: AI system intended to interact directly with natural persons and generating synthetic content (study plans), subject to transparency obligations under AI Act Article 50(1) and 50(2)

9.4 Application Functionality

Instructions for Use for Deployers:

- System requires Azure AI Foundry environment setup and API key configuration.
- System requires a Serpes API key to be able to search the topics to be studied on the web.
- Learning plans are delivered via web interface or downloadable formats (.md extension).

Model Capabilities:

- Analyzes user profiles (role, experience level, learning goals) to generate personalized study paths.
- Creates structured learning guides with web resources and academic paper references.
- Generates study calendars with timeline recommendations.
- Generates a flowchart that guides the user through the individual topics to be learned in order.
- Limitations: Content quality depends on input specificity; may require manual review for highly specialized domains.
- Languages supported: English

Input Data Requirements:

- User profile: a user prompt with role description, experience level, learning objectives.
- Valid inputs: a user prompt with clear role definitions, specific learning goals, realistic timeframes.
- Invalid inputs: a user prompt with vague objectives, undefined roles, unrealistic timeline constraints.

Output Explanation:

- Generates a comprehensive study plan including topic sequences, resource links, and scheduling.
- Provides structured learning paths with difficulty progression and estimated timeframes.
- Outputs include web resources, academic references, and visual flowcharts for study progression.

System Architecture Overview:

- CrewAI-based multi-agent orchestration with six specialized crews.
- Azure AI Foundry deployment utilizing GPT-4.1 models.
- Web-based interface Streamlit-based for user interaction and plan delivery.

9.5 Models and Datasets

9.6 Models

Model	Documentation	Description of Application Usage
GTP-4.1	GPT-4.1 Documentation	Primary language model used across CrewAI crews for natural language processing and content generation

9.7 Datasets

No custom datasets are utilized.

Note, the system primarily relies on foundation models (GPT-4.1) rather than custom-trained models. Additional datasets may be used for fine-tuning content relevance and domain-specific knowledge validation. See Azure AI Foundry and CrewAI documentation for detailed model specifications and capabilities.

9.8 Deployment

Cloud Setup

- **Cloud Provider**: Microsoft Azure
- **Region**: [Specify your Azure region]
- **Required Services**: - Azure AI Foundry (GPT-4.1 model hosting) - Azure Compute (for application hosting)
- **Resource Configurations**: - Standard compute instances for Streamlit application - API-based model access (no dedicated GPU/TPU requirements)
- **Network Setup**: Standard Azure networking with secure API endpoints

APIs

- **Azure AI Foundry API**: - Authentication: API key-based - Endpoints: GPT-4.1 model inference
- **SerperDevTool API**: - Authentication: API key - Purpose: Web search for educational content - Rate limits: As per provider specifications
- **arXiv API**: - Authentication: Public API (no key required) - Purpose: Academic paper retrieval - Rate limits: Standard arXiv query limits

9.9 Integration with External Systems

Systems Dependencies

- **External APIs**: SerperDevTool, arXiv, Azure AI Foundry
- **Local Services**: MLFlow server (localhost:5000)
- **Programming Language**: Python
- **Framework**: CrewAI
- **UI Framework**: Streamlit

Data Flow

- User input → Input Crew → Sequential crew processing → Final output delivery

- No persistent database storage (session-based data handling)

Error Handling

- API timeout handling for external services
- Retry mechanisms for failed crew tasks
- Fallback content generation for unavailable resources

9.10 Deployment Plan

Infrastructure

- **Environments**: Development (local), Production (Azure)
- **Scaling**: Manual scaling based on usage patterns
- **Backup**: Code repository backup (no data persistence required)

Integration Steps

1. Azure AI Foundry model deployment and API configuration
2. External API key configuration (SerperDevTool)
3. CrewAI framework setup with specialized crews
4. Streamlit application deployment
5. MLFlow monitoring server setup (local environment)

Dependencies

- **Python Libraries**: CrewAI, Streamlit, arxiv, MLFlow
- **External Services**: Azure AI Foundry, SerperDevTool API
- **Runtime Requirements**: Python 3.8+, internet connectivity for API access

Rollback Strategy

- Version-controlled deployment with Git
- Quick rollback to previous stable release
- API key rotation procedures for security incidents

9.11 Lifecycle Management

Performance Monitoring

- **MLFlow Integration**: Local server (localhost:5000) for real-time agent performance tracking
- **Response Time Monitoring**: End-to-end learning plan generation latency
- **API Health Checks**: Azure AI Foundry, SerperDevTool, and arXiv API availability
- **Error Rate Tracking**: Failed crew executions and API timeouts

Ethical Compliance Monitoring

- **Content Quality Assessment**: Regular review of generated learning plans for appropriateness
- **Bias Detection**: Monitoring for discriminatory patterns in role-based recommendations
- **Transparency Compliance**: Ensuring AI Act Article 50 disclosure requirements are met

- **Data Privacy**: Monitoring adherence to GDPR requirements for user profile data

Versioning and Change Logs

See the application on the GitHub Repository

9.12 Metrics and KPIs

Execution Performance

- **Crew Execution Time**: Individual crew processing time in seconds, measured from workflow start to crew completion
- **Total Workflow Time**: End-to-end learning plan generation time from user input to final study plan
- **Total Tokens**: Complete token usage per crew execution
- **Prompt Tokens**: Input processing token count
- **Completion Tokens**: Output generation token count
- **Output Length**: Character count of generated content per crew to assess comprehensiveness

System Reliability

- **Workflow Status**: Success/failure tracking with error message logging
- **Crew Success Rate**: Percentage of successful crew executions without errors
- **Input Validation**: Detection and handling of invalid inputs with escape sequence protection

Custom Quality Metrics

- **Study Plan Quality Score**: PlanningCrew output evaluation (0.0–10.0 scale)
- **Excellent Plan**: 10.0 (comprehensive, well-structured, realistic)
- **Good Plan**: 7.5 (solid but may need optimization)
- **Poor Plan**: 2.5 (inadequate or unrealistic)
- **Search Relevance Score**: WebCrew resource quality assessment (0.0–10.0 scale)
- **Highly Relevant**: 10.0 (directly addresses learning needs)
- **Somewhat Relevant**: 5.0 (generally related but lacking depth)
- **Not Relevant**: 0.0 (off-topic or inadequate)

MLflow GenAI Evaluation

- **Relevance to Query**: Final study plan relevance to original user question using MLflow's RelevanceToQuery scorer
- **Quality Feedback**: Detailed rationale for quality scores

Crew-Specific Tracking

- **Crew Type**: Individual crew identification
- **Number of Agents**: Agent count per crew
- **Output Type**: Classification of crew output format
- **Token Usage Details**: Comprehensive token consumption breakdown

Input/Output Analysis

- **Input Question**: Original user request logging

- **Sanitized Input**: Processed and validated input
- **Workflow Type**: Classification as EY Junior Accelerator workflow

Continuous Monitoring

- **Real-world Usage Analysis**: Monitor actual learning plan utilization
- **Performance Drift Detection**: Identify degradation in content quality or relevance
- **API Dependency Health**: Track external service reliability
- **User Feedback Integration**: Incorporate feedback for system improvements

Maintenance Tasks

- **Periodic Content Validation**: Review and update resource recommendations
- **API Integration Updates**: Maintain compatibility with external services
- **Security Reviews**: Regular assessment of data handling and API security
- **Compliance Audits**: Ensure ongoing Article 50 and GDPR compliance

Monitoring Logs

- **MLFlow Dashboards**: Real-time agent performance data

9.13 Risk Management System

Assessment Process

- **Risk Identification**: Systematic review of potential failure modes across all CrewAI crews
- **Impact Analysis**: Assessment of consequences on users, organizations, and compliance

High Priority Risks

Biased Learning Recommendations

- **Potential Harmful Outcome**: Discriminatory learning paths based on role, background, or protected characteristics
- **Likelihood**: Medium – GPT-4.1 may exhibit inherent biases in content generation
- **Severity**: High – Could lead to unfair career development opportunities and legal compliance issues

Privacy and Data Handling Breaches

- **Potential Harmful Outcome**: Unauthorized access to user profile information or learning preferences
- **Likelihood**: Low – Limited data persistence, but API communications present risks
- **Severity**: High – GDPR violations and personal data compromise

Medium Priority Risks

Inappropriate Content Generation

- **Potential Harmful Outcome**: Recommendation of irrelevant, outdated, or harmful learning resources
- **Likelihood**: Medium – Dependent on external API content quality and model hallucination
- **Severity**: Medium – Could waste time and provide poor learning outcomes

External API Dependencies Failure

- **Potential Harmful Outcome**: System unavailability or incomplete learning plan generation

- **Likelihood**: Medium – Dependent on third-party service reliability
- **Severity**: Medium – Service disruption and user experience degradation

Transparency Compliance Failure

- **Potential Harmful Outcome**: Non-compliance with EU AI Act Article 50 disclosure requirements
- **Likelihood**: Low – Implementation-dependent
- **Severity**: Medium – Regulatory non-compliance and potential penalties

Low Priority Risks

Content Quality Degradation

- **Potential Harmful Outcome**: Decreased relevance of academic papers and web resources over time
- **Likelihood**: High – Natural content aging and changing field landscapes
- **Severity**: Low – Gradual decrease in learning effectiveness

Performance Degradation

- **Potential Harmful Outcome**: Slow response times affecting user experience
- **Likelihood**: Medium – Dependent on API latency and system load
- **Severity**: Low – User convenience impact only

Risk Mitigation Measures

Bias Prevention

- **Content Validation**: Regular review of generated learning plans for discriminatory patterns
- **Diverse Testing**: Validation across different roles, experience levels, and demographics
- **Prompt Engineering**: Carefully designed crew instructions to minimize biased outputs
- **Monitoring Protocols**: MLFlow tracking for bias detection metrics

Data Protection

- **Minimal Data Collection**: Only essential user profile information processed
- **Session-Based Storage**: No persistent storage of personal data
- **API Security**: Secure handling of authentication credentials and user data in transit
- **Access Controls**: Restricted system access and audit logging

Content Quality Assurance

- **Multi-Source Validation**: Cross-referencing recommendations across different crews
- **Resource Filtering**: Quality checks for web resources and academic papers
- **Regular Content Updates**: Periodic validation of resource relevance and accuracy
- **User Feedback Integration**: Mechanisms for quality improvement based on user input

Compliance Assurance

- **Transparency Implementation**: Clear AI system disclosure in user interface
- **Documentation Maintenance**: Comprehensive record-keeping for audit purposes
- **Regular Compliance Reviews**: Periodic assessment against EU AI Act requirements

9.14 Human Oversight

Human-in-the-loop

HR managers and training coordinators can review and modify generated learning plans before delivery, with manual verification of recommended resources and academic papers for relevance and appropriateness, while human decision-makers retain final authority over learning path implementation and employee assignments.

Training Requirements

HR staff responsible for system operation are trained on learning plan review and validation procedures, new hires receive orientation on interpreting AI-generated learning plans, and staff are educated on EU AI Act Article 50 transparency requirements and GDPR obligations.

System Limitations

The system cannot assess highly specialized or emerging technical domains without subject matter expert review, has limited ability to account for personal learning preferences and accessibility needs, may generate outdated content requiring periodic human validation, and may not fully account for organization-specific learning culture and practices.

9.15 Incident Management

Common Issues

- **Resource Unavailability**: Broken links or inaccessible academic papers in generated plans
- **Content Misalignment**: Recommendations not matching actual job requirements or organizational standards
- **API Service Disruptions**: Temporary failures of SerperDevTool or arXiv services affecting content generation
- **Performance Degradation**: Slow response times during peak usage periods

Support Contact

- **Internal IT Support**: Standard business hours technical assistance for system issues
- **HR Department**: Primary contact for content quality concerns and learning plan modifications
- **System Administrator**: Escalation contact for serious technical failures or compliance issues

Troubleshooting

This section outlines potential issues that can arise during the deployment of LearningPath AI, along with their causes, resolutions, and best practices for mitigation.

Insufficient Resources

- **Problem**: Azure AI Foundry API rate limiting or quota exhaustion during peak usage periods leading to failed crew executions and incomplete learning plan generation.
- **Mitigation Strategy**: Monitor API usage through Azure dashboards, implement request queuing mechanisms, and establish usage alerts to prevent quota exhaustion.

Network Failures

- **Problem**: Network connectivity issues preventing access to external APIs (SerperDevTool, arXiv, Azure AI Foundry) causing crew failures and incomplete content generation.
- **Mitigation Strategy**: Implement retry mechanisms with exponential backoff, configure timeout settings, and establish fallback procedures for critical API dependencies.

API Failures

- **Problem**: External APIs (SerperDevTool, arXiv, Azure AI Foundry) are unreachable due to service outages, authentication failures, or rate limiting.

- **Mitigation Strategy**: Implement comprehensive error handling and API health checks.

Data Format Mismatches

- **Problem**: Unexpected response formats from external APIs causing crew processing failures and incomplete learning plan generation.
- **Mitigation Strategy**: Implement robust data validation and parsing, handle API response variations gracefully, and maintain fallback processing for malformed responses.

Performance or Deployment Issues

- **Problem**: Inconsistent or poor-quality learning plan generation due to GPT-4.1 model variations, prompt engineering issues, or crew configuration problems.
- **Mitigation Strategy**: Monitor output quality through MLFlow tracking, implement content validation checks, maintain versioned prompt templates, and establish quality review processes.

Safety and Security Issues

- **Problem**: API keys or Azure AI Foundry credentials are exposed or compromised, leading to unauthorized system access or resource consumption.
- **Mitigation Strategy**: Secure credential storage using environment variables or Azure Key Vault, implement API key rotation procedures, and monitor for unusual usage patterns.

Data Breaches

- **Problem**: User profile information or learning preferences are exposed due to insecure handling in crew processing or Streamlit interface.
- **Mitigation Strategy**: Minimize data retention, implement secure session management, ensure HTTPS communication, and avoid logging sensitive user information.

Monitoring and Logging Failures

- **Problem**: Insufficient visibility into crew execution failures, API errors, or system performance issues through MLFlow monitoring.
- **Mitigation Strategy**: Enhance logging coverage across all crews, implement structured logging with appropriate detail levels, and establish monitoring dashboards for critical system metrics.

9.16 Documentation Authors

Team 1

- **Emanuele Addari**
- **Michele Bruno**
- **Anna Setzu**
- **Giosuè Sglavo**

CHAPTER TEN

INDICES AND TABLES

- genindex
- modindex
- search