# Delivery Instructions

## 1. Synthetic Data Generation

### 1.1. Conversation Flow Schema

This project includes an implementation for a metalanguage that allows the definition of conversation flow schemas.

More information about the schema can be found in the `docs/conversation-flow-schema.pdf` file.

The data generation pipeline looks for a defined schema in the `<project_root>/config` directory. The schema is defined in the `conversation_flow.yml` file.

> **Note**
>
> Upon execution, the project will generate all possible unique conversation flows based on the defined schema (taking into account the number of steps, the type of steps, and the possible transitions between steps).

### 1.2. Agent and Candidate Personas, Job Openings

The data generation pipeline uses the concept of personas and job openings to generate synthetic conversations between HR agents and potential candidates, discussing a specific job opening.

The personas and job openings are defined in the `<project_root>/config` directory. Personas and job openings are defined in the `seeds.yml` file.

- `agent`: HR agent personas (minimum of `1`). List of:
  - `first_name`: HR agent first name.
  - `last_name`: HR agent last name.
  - `company_name`: company name.
  - `company_info`: company additional information (description, industry, etc.).
  - `career_site_url`: company's career site URL.
- `candidates`: potential candidate profiles (minimum of `1`). List of:
  - `first_name`: candidate first name.
  - `last_name`: candidate last name.
  - `experiences`: candidate professional experiences (minimum of `1`). List of:
    - `company`: company name.
    - `position`: position held.
    - `description`: job description.
- `job_openings`: job openings (minimum of `1`). List of:
  - `title`: job opening title.
  - `description`: job opening description.
  - `requirements`: job opening requirements.

> **Warning**

> The number of HR agent personas, candidate personas, and job openings will directly impact the number of synthetic conversations generated. Ensure that the number of defined personas and job openings is appropriate for the desired number of synthetic conversations.

## 1.3. Seeds

The data generation pipeline uses the concept of a seed in order to generate a synthetic conversation.

Each seed is a combination of:

- **HR agent persona**: the persona of the HR agent in the conversation.
- **Candidate persona**: potential candidate profile.
- **Job opening**: job opening details.
- **Conversation flow**: specific sequence of conversation steps between the HR agent and the candidate.

The seed data is defined in the `<project_root>/config` directory. The seed data is defined in the `seeds.yml` and `conversation_flow.yml` files, as described in the previous sections.

> **Note**
>
> Upon execution of the pipeline, all possible combinations of seeds are generated, and a synthetic conversation is generated for each seed.
>
> Mathematically, the number of synthetic conversations generated is equal to the cross product of HR agent personas, candidate personas, job openings, and conversation flows.
>
> $$\text{Seeds} = (\text{HR agent personas}) \times (\text{Candidates}) \times (\text{Job openings}) \times (\text{Conversation flows})$$

## 1.4. Prompt Templates

The data generation pipeline uses prompt templates to generate the multi-turn conversation prompts for the synthetic conversations.

The prompt templates are defined in the `<project_root>/config` directory, at the `templates.yml` file.

There are three main types of prompt templates used in the data generation pipeline:

- `base`: base prompt template for the conversation, shared by both the HR agent and the candidate personas.
- `agent`: specific prompt instructions for the HR agent persona.
- `candidate`: specific prompt instructions for the candidate persona.
- `fine_tuning`: specific prompt template used in the final fine-tuned model.

> **Note**
>
> The prompt templates are different due to the different roles of the HR agent and the candidate in the conversation. Additionally, context-specific information is included in the prompt templates to generate more realistic conversations depending on the current conversation stage.

### 1.5. Data Generation Pipeline

The data generation pipeline is implemented in the `<project_root>/src` directory.

In order to execute the synthetic data generation pipeline.

1. `cd` into the `<project_root>/src/` directory.
2. Set the right environment variables in a `.env` file. Use the `.env.example` file as a template.
    - `OPENAI_API_KEY`: OpenAI API key. Required for generating synthetic data using OpenAI's `GPT-4o-mini` model.
3. Make any changes needed to the `<project_root>/config` directory.
4. Create and activate a virtual environment. Install the required dependencies.

```
python -m venv venv
source venv/bin/activate
pip install -r requirements-data-gen.txt
```

5. Run the data generation pipeline.

```
python -m generate_dataset.py
```

6. The generated data will be saved in the `<project_root>/output` directory as a `JSONL` file.

## 2. Model Fine-Tuning

This project includes an implementation for a transformer-based model fine-tuning pipeline.

The fine-tuning pipeline is powered by `unsloth.ai`, a library that provides a high-level API for performing a parameter-efficient fine-tuning of transformer-based models.

### 2.1. Model Fine-Tuning Pipeline

The model fine-tuning pipeline is implemented in the `<project_root>/src` directory.

In order to execute the model fine-tuning pipeline.

1. `cd` into the `<project_root>/src/` directory.
2. Set the right environment variables in a `.env` file. Use the `.env.example` file as a template.
    - `HF_TOKEN`: Hugging Face API token. Required for uploading the fine-tuned model to the Hugging Face model repository.
    - `BASE_MODEL`: Hugging Face repository name for the base model to be fine-tuned. Pipeline expects the model to be available in the Hugging Face model repository (either public or private).
    - `HF_DATASET`: Hugging Face dataset name for the fine-tuning data. Pipeline expects the dataset to be available in the Hugging Face datasets repository (either public or private).
    - `MODEL_NAME`: name of the output model, result of the fine-tuning process.

    - `HF_MODEL_NAME`: Hugging Face repository name for the output model. Pipeline will upload the fine-tuned model to the Hugging Face model repository (either public or private).
3. If needed, make any changes to the fine-tuning configuration in the `<project_root>/src/fine_tune.py` file.
4. Create and activate a virtual environment. Install the required dependencies.

```
python -m venv venv
source venv/bin/activate
pip install -r requirements-fine-tuning.txt
```

5. Run the model fine-tuning pipeline.

```
python -m fine_tune.py
```

6. The fine-tuned model will be saved in the `<project_root>/output` directory as a `tar.gz` file. Additionally, the fine-tuned model will be uploaded to the Hugging Face model repository.

> **Note**
>
> By default, the pipeline will quantize the model to `4-bit` precision. This can be changed in the `<project_root>/src/fine_tune.py` file.

> **Tip**
>
> More information about the `unsloth.ai` library can be found in the official documentation.

## 3. Model Registration

This project includes an implementation for registering the fine-tuned model in the MLFlow model registry.

The registered model consists of a custom MLFlow `pyfunc` wrapper that allows the model to be used for inference.

Once registered, the model can be pulled from the MLFlow model registry.

### 3.1. Model Registration Configuration

The model registration pipeline looks for a configuration file in the `<project_root>/config` directory. The configuration file is defined in the `inference.yml` file.

The following configuration parameters are required:

- `model`:
    - `id`: name of the model to be registered in the MLFlow model registry.
    - `config`:
        - `hf_model_id`: Hugging Face repository name for the fine-tuned model.
        - the rest of parameters are model-specific and depend on the model architecture. Check Hugging Face Transformers model documentation for more information.

- `prompt`: base instruction prompt for the model
- `prompt_data`: values for the fixed variables to be used in the instruction prompt
- `model_signature`: inference model signature for the MLFlow model registry
- `input_examples`: inference input examples for the MLFlow model registry

## 3.2. Model Registration Pipeline

The model registration pipeline is implemented in the `<project_root>/inference` directory.

In order to execute the model registration pipeline.

1. `cd` into the `<project_root>/inference/` directory.
2. Set the right environment variables in a `.env` file. Use the `.env.example` file as a template.
    - `MLFLOW_TRACKING_URI`: MLFlow tracking URI. Required for connecting to the MLFlow tracking server.
    - `MLFLOW_TRACKING_USERNAME`: MLFlow tracking username. Required for connecting to the MLFlow tracking server.
    - `MLFLOW_TRACKING_PASSWORD`: MLFlow tracking password. Required for connecting to the MLFlow tracking server.
    - `HF_MODEL_NAME`: Hugging Face repository name for the fine-tuned model. Required for pulling the fine-tuned model from the Hugging Face model repository. `HF_TOKEN`: Hugging Face API token. Required for pulling the fine-tuned model from the Hugging Face model repository.
3. If needed, make any changes to the model registration configuration in the `<project_root>/config/inference.yml` file.
4. Create and activate a virtual environment. Install the required dependencies.

    ```
    python -m venv venv
    source venv/bin/activate
    pip install -r requirements.txt
    ```

5. Run the model registration pipeline.

    ```
    python -m register.py
    ```

## 3.3. Local Model Inference

The inference pipeline has a local inference script that allows for testing the fine-tuned model locally.

In order to execute the local inference pipeline.

1. Follow the steps in the `3.1. Model Registration Pipeline` section.
2. Modify the `<project_root>/inference/input_test.yaml` file with the desired input example to test the model.
3. Run the local inference pipeline.

```
python -m test.py
```

4. The output of the model inference will be displayed in the console.

## 3.4. Model Deployment

The fine-tuned model has been deployed, and is available for inference via the project's API deployed in Kubernetes.

Model inference is executed in a RunPod serverless endpoint. Invocations are controlled by the backend API, and executed asynchronously in RunPod.

The API endpoint is available at `https://chat.toby.nieve.ai/`.

OpenAPI documentation for the API is available at `https://chat.toby.nieve.ai/docs`. Please, refer to this documentation for additional information on the available endpoints and their usage, as well as the expected input and output schemas for all mentioned endpoints.

In order to test the model inference from the project's backend.

1. Create a candidate profile using the `POST /profiles` endpoint.
2. Create a job opening using the `POST /job_openings` endpoint.
3. Create a conversation between the HR agent and the candidate using the `POST /conversations` endpoint.
4. Create the initial message/s using the `POST /conversations/{conversation_id}/messages/` endpoint.
5. Use the `POST /conversations/{conversation_id}/chat` endpoint to interact with the model and generate the next message in the conversation.

> **Tip**
>
> If you don't want to automatically persist the generated message in the conversation DB, you can use the `log = False` parameter in the request body. This is useful for testing the model without persisting the generated messages and having to recreate new conversations from scratch.