Dorian ROLLAND

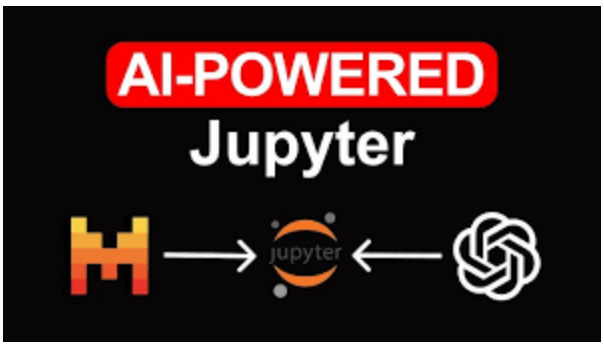
Connect Jupyter AI and a local LLM inside a Docker container

Cea Grenoble

Introduction

Jupyter AI is an extension for JupyterLab that enables seamless integration with various AI models, including OpenAI's ChatGPT, Mistral, and Google's Gemini.While Jupyter AI is not an AI model itself, it acts as a bridge allowing users to interact with AI models directly within Jupyter notebooks.

For security, privacy, or offline usage, it can be beneficial to run a local AI model instead of relying on cloud-based services. This report details the process of setting up Jupyter AI inside a Docker container along with a local LLM using Ollama.



Installing Jupyter AI

To install Jupyter AI, use the following command:

```
pip install 'jupyter-ai[all]'
```

After installation, API keys are required to connect to external AI providers, but for this setup, we will use a local model.

To verify the installation, launch JupyterLab and check if the Jupyter AI extension is available:

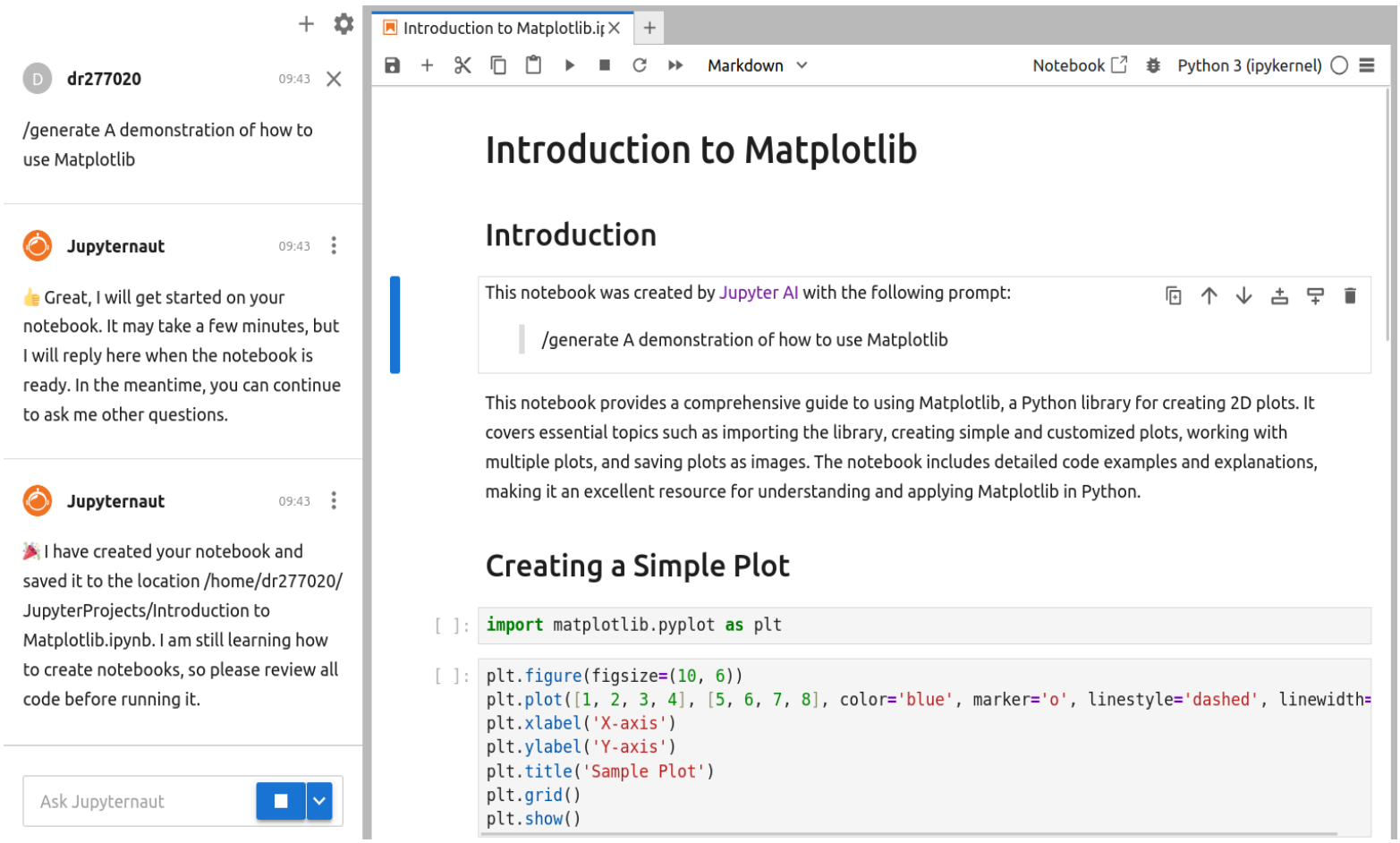
```
jupyter labextension list
```

```
gre057597:~% jupyter labextension list
`sys_prefix` level settings are read-only, using `user` level for migration to `lockedExtensions`
JupyterLab v4.3.5
/home/dr277020/.local/share/jupyter/labextensions
        jupyterlab_pygments v0.3.0 enabled OK (python, jupyterlab_pygments)
        @jupyter-widgets/jupyterlab-manager v5.0.13 enabled OK (python, jupyterlab_widgets)
        @jupyter-ai/core v2.29.1 enabled OK (python, jupyter_ai)
        @jupyter-notebook/lab-extension v7.3.2 enabled OK
```

Jupyter AI Functionalities

Jupyter AI provides a chat interface and powerful commands within notebooks, enhancing productivity by allowing users to interact directly with AI models without leaving the notebook environment. Below are some of the key functionalities that Jupyter AI offers:

- `/ask` - Ask a question to the AI and receive an instant response.
- `/generate` - Generate text-based content such as summaries, reports, or creative writing pieces. Here is an example of what you can do with /generate.



- `/learn` - Train AI on specific documents and reference materials, supporting various file extensions like `.py`, `.md`, `.ipynb`, `.html`, and `.pdf`.
- `/export` - Export conversations and AI-generated content for further use or documentation.
- `/fix` - Detect and suggest corrections for code snippets or markdown text.
- `/help` - Display available commands and their usage for reference.

AI Magic Commands

Jupyter AI also supports AI magic commands that allow direct AI interactions within a notebook cell.

The `%%ai` cell magic allows you to invoke a language model of your choice with a given prompt. The model is identified with a **global model ID**, which is a string with the syntax `<provider-id>:<local-model-id>`, where `<provider-id>` is the ID of the provider and `<local-model-id>` is the ID of the model scoped to that provider. The prompt begins on the second line of the cell.

For example, to send a text prompt to the provider `google gemini` and the model ID `gemini-1.0-pro`, enter the following code into a cell and run it:

```
%%ai gemini-1.0-pro

Explain the concept of parallel computing and give an example.
```

This command sends the request to the gemini model and retrieves an AI-generated explanation.

Practical Use Cases

```
%load_ext jupyter_ai_magics
```

```
The jupyter_ai_magics extension is already loaded. To reload it, use:
  %reload_ext jupyter_ai_magics
```

```
%%ai gemini-1.0-pro
Explain the concept of parallel computing and give an example.
```

Parallel Computing

Parallel computing is a type of computing that utilizes multiple processing units (CPUs or GPUs) to perform computations simultaneously. It is used to solve complex problems that require significant computational resources and can significantly reduce the time required to complete tasks.

Concept:

Parallel computing involves dividing a problem into smaller subtasks that can be executed concurrently. These subtasks are distributed across multiple processing units, which work independently to solve their assigned portions. Once all subtasks are completed, the results are combined to produce the final solution.

Jupyter AI extends beyond simple chat interactions by supporting deeper integration into workflows. Some practical applications include:

- **Code Assistance:**

```
%%ai gemini-1.0-pro
Optimize the following Python function for better performance.
```

- **Data Analysis:**

```
%%ai gemini-1.0-pro
Analyze this dataset and provide insights on potential correlations.
```

- **Documentation Generation:**

```
%%ai gemini-1.0-pro
Generate documentation for this Python class with detailed
explanations.
```

- **Debugging Assistance:**

```
%%ai gemini-1.0-pro
Find and explain the bug in this piece of code.
```

By leveraging these functionalities, Jupyter AI becomes a powerful tool for developers, data scientists, and researchers to streamline workflows, enhance productivity, and integrate AI-powered insights directly into their coding environment. Jupyter AI provides a chat interface and powerful commands within notebooks

Setting Up a Local LLM with Jupyter AI inside Docker

Why Use a Local LLM?

Running a local LLM has multiple benefits:

- No dependency on external API providers.
- Improved data privacy.
- Reduced operational costs.
- No internet requirement after initial setup.

Dockerfile for Jupyter AI and Local LLM

Below is the Dockerfile to set up a container with Jupyter AI and a local LLM using Ollama:

```
# Use Ubuntu as base image
FROM ubuntu:latest

# Install dependencies
RUN apt-get update && apt-get install -y \
    vim \
    python3 \
    python3-pip \
    python3-venv \
    fontconfig \
    curl \
    wget \
    xdg-utils

# Install Ollama (for local LLM management)
RUN curl -fsSL https://ollama.com/install.sh | sh

# Add Ollama to PATH
ENV PATH="/root/.ollama/bin:$PATH"


# Start Ollama and download the Mistral model
RUN ollama serve & \
    sleep 5 && \
    ollama pull mistral

# Create Python virtual environment and install JupyterLab and Jupyter AI
RUN python3 -m venv /opt/venv && \
    /opt/venv/bin/pip install --upgrade pip && \
    /opt/venv/bin/pip install \
        jupyterlab \
        'jupyter-ai[all]'

# Add virtual environment to PATH
ENV PATH="/opt/venv/bin:$PATH"

# Set working directory
WORKDIR /home

# Copy startup script
COPY start-jupyter.sh .
RUN chmod +x start-jupyter.sh

# Expose necessary ports
EXPOSE 11434 8888

# Start JupyterLab
ENTRYPOINT ["/bin/bash", "/home/start-jupyter.sh"]
```

Below start-jupyter2.sh

```
#!/bin/bash

# Démarrer Ollama en arrière-plan
ollama serve &

# Attendre que Ollama soit prêt
sleep 5

# Démarrer JupyterLab
jupyter lab --ip=0.0.0.0 --port=8888 --allow-root \
```

The `start-jupyter2.sh` script is responsible for initializing the local LLM (Ollama) and JupyterLab within the Docker container. Its role is as follows:

1. Starts Ollama in the background (`ollama serve &`) → This launches the local AI model manager, which allows Jupyter AI to interact with an on-device LLM.
2. Waits for Ollama to be ready (`sleep 5`) → Since Ollama takes a few seconds to initialize, a short delay ensures that the service is fully available before proceeding.
3. Launches JupyterLab (`jupyter lab --ip=0.0.0.0 --port=8888 --allow-root`) → This command starts JupyterLab, making it accessible from any network interface inside the container.

This script is essential because it ensures that both Ollama and JupyterLab start correctly, allowing seamless interaction between Jupyter AI and the local LLM.

Running the Docker Container

To make it simpler, you can directly download my project from my Docker Hub.

To build and run the container:

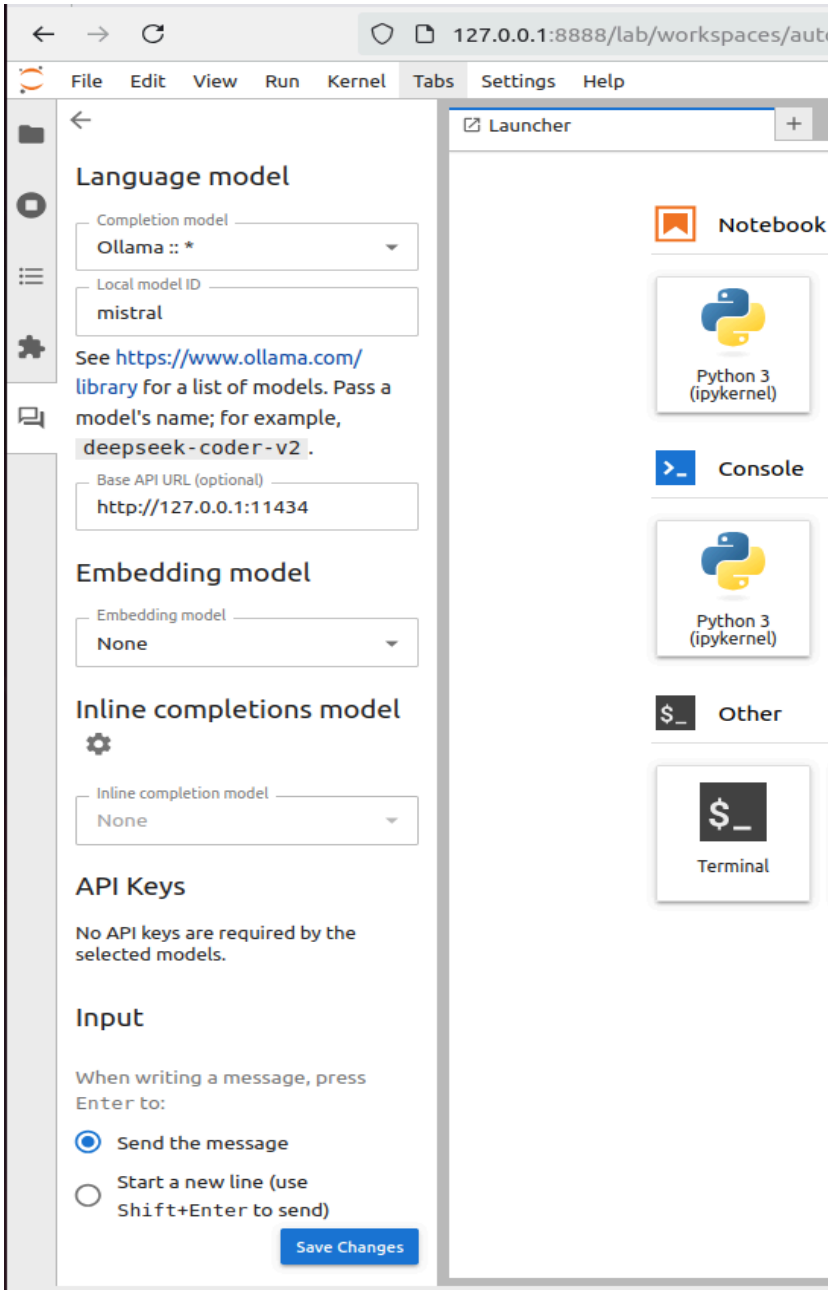
```
docker pull dorianalp38/jupyter-ai:latest

docker run --rm -it -p 11434:11434 -p 8888:8888 dorianalp38/jupyter-ai:latest
```

JupyterLab will be accessible at: `http://127.0.0.1:8888`

Testing the Local AI Model

Now you need to set up Jupyter AI to connect it to your local AI. Here is an example.



To verify that Mistral is installed and working correctly, you can test it with a prompt.

Conclusion

This setup provides a local, secure, and cost-effective AI environment within JupyterLab. By leveraging Docker, we ensure a portable and reproducible configuration that does not rely on external AI service providers.

Future improvements could include:

- Integrating additional models (e.g., Llama 3, Falcon) using Ollama.
- Optimizing performance by running the container on GPU-enabled hardware.
- Implementing a local AI trained using Retrieval-Augmented Generation (RAG) on tools used in our lab, such as RemoteManager. This presents challenges in integrating domain-specific data efficiently while ensuring real-time retrieval and adaptation to evolving datasets.

By using a Dockerized Jupyter AI instance with a local LLM, we achieve an efficient, scalable, and privacy-friendly solution for AI-powered workflows.

For more details, visit the GitHub repository: https://github.com/Dorianrolland/Jupyter-AI.