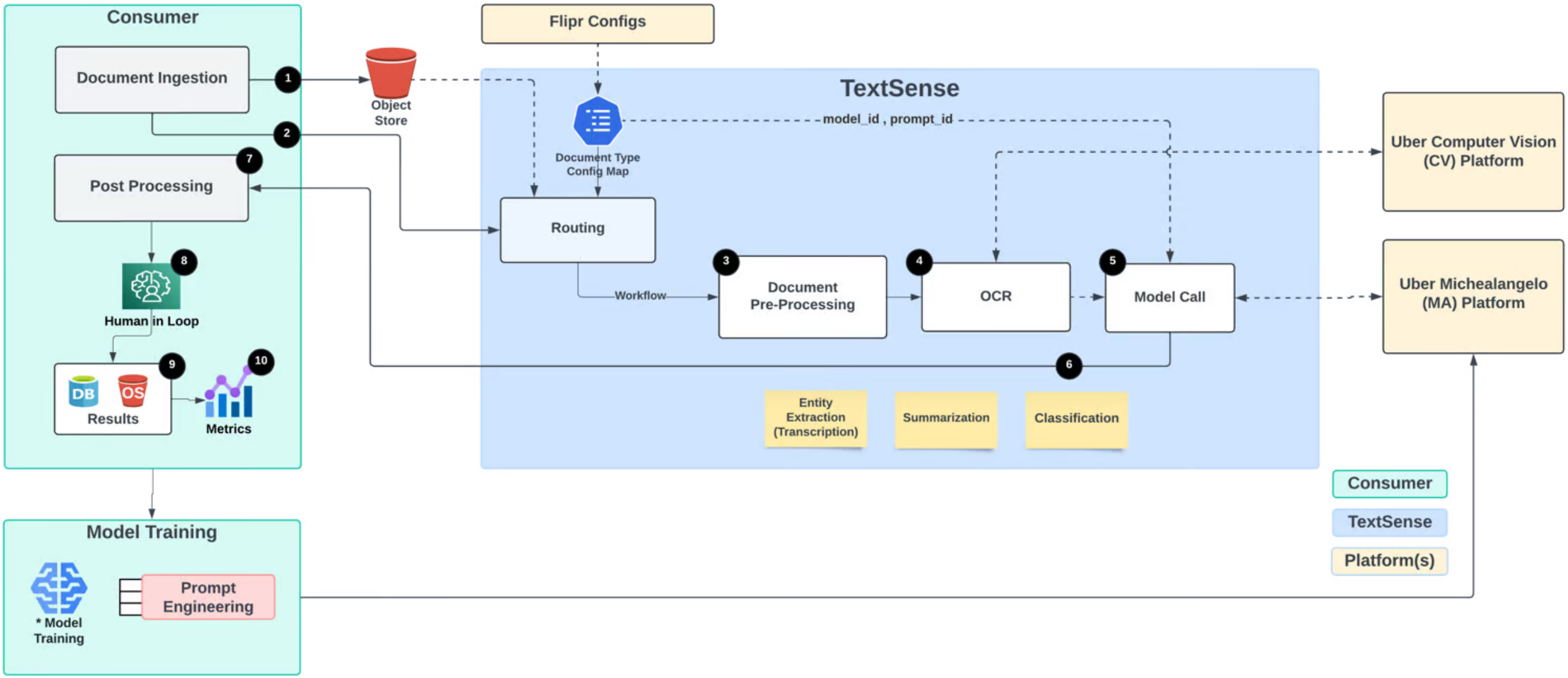
CS 696 Applied Large Language Models
Spring Semester, 2025
Doc 24 MLOps
Apr 24, 2025

Advancing Invoice Document Processing at Uber

https://www.uber.com/en-IL/blog/advancing-invoice-document-processing-using-genai/



why i wont be vibe coding anymore

https://varunraghu.com/why-i-wont-be-vibe-coding-anymore/

i realized i hadn’t learnt a new concept in weeks

that’s when it struck me. coding isn’t about the finished product. its a lot like writing. its about the process. its about how you approach a problem. its critical thinking.

and i don’t think i’m ready to let ai take these away from me. i’m going back to writing shitty code, slowly and deliberately.

Issues with Building ML Systems

Ingest, clean, and validate fresh data

Training versus inference setups

Compute and serve features in the right environment

Serve the model in a cost-effective way

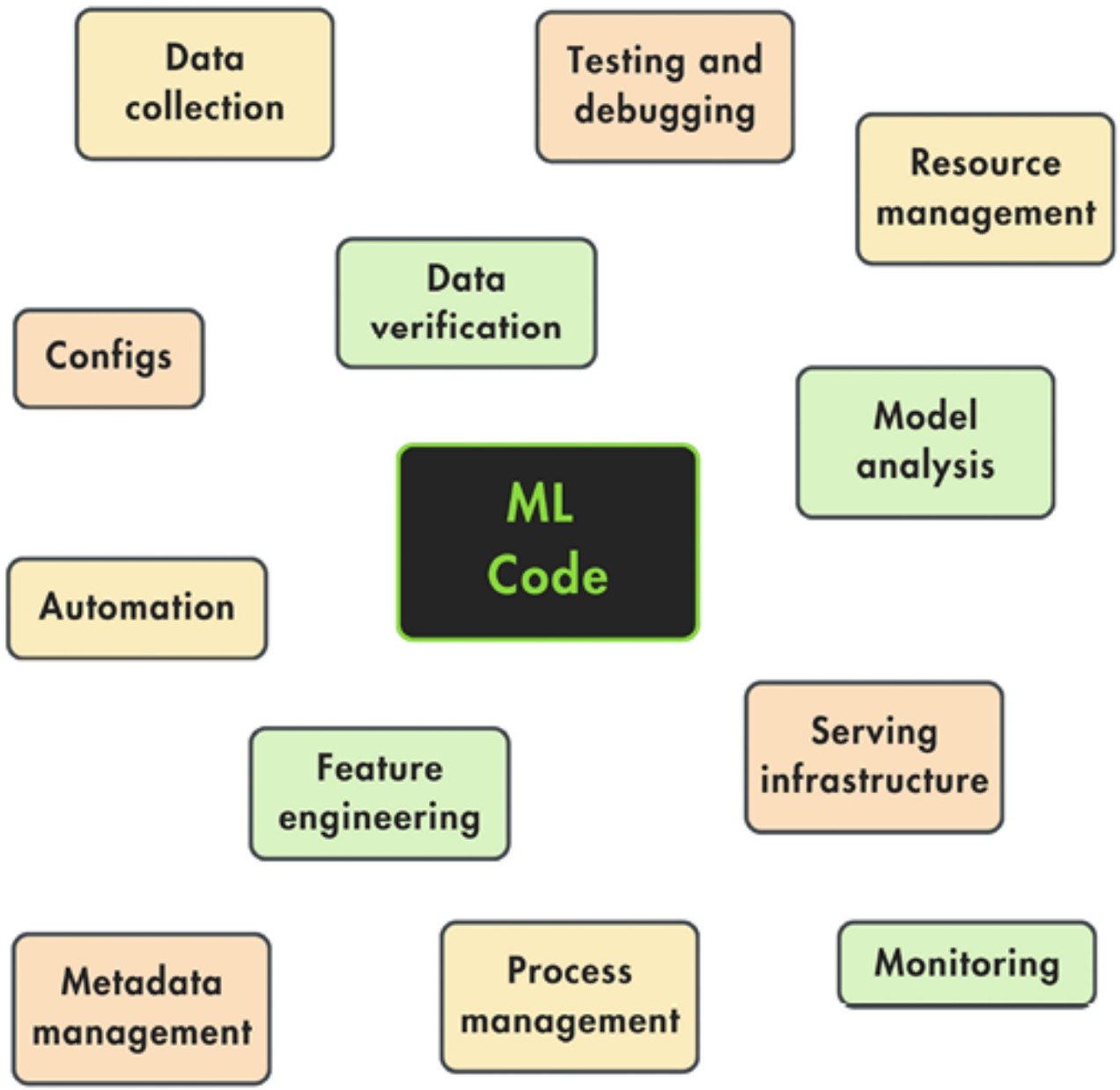
Version, track, and share the datasets and models

Monitor your infrastructure and models

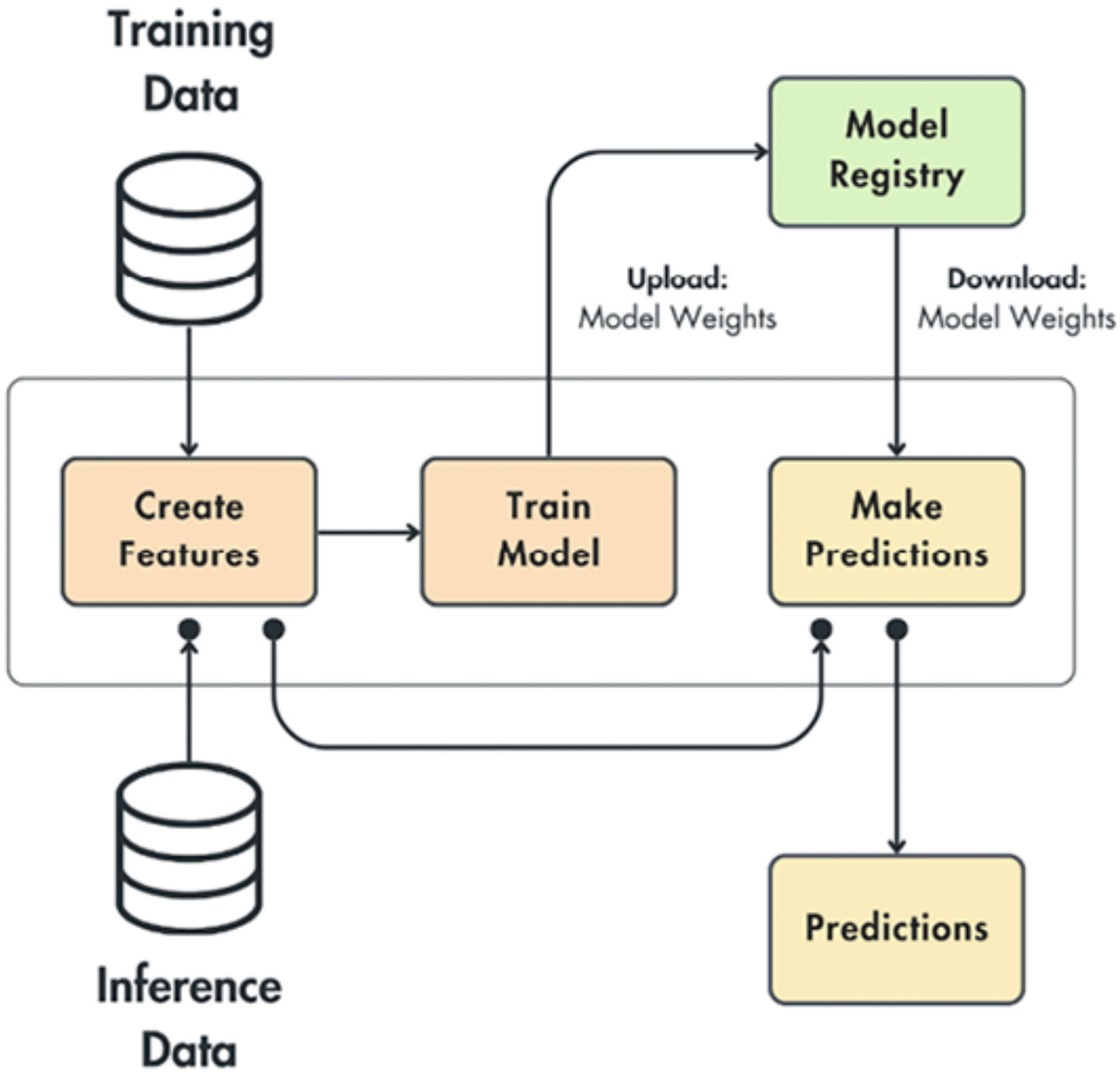
Deploy the model on a scalable infrastructure

Automate the deployments and training

LLM Engineer’s Handbook, Lustin & Labonne


Data collection
Testing and debugging
Resource management
Data verification
Configs
Model analysis
ML Code
Automation
Feature engineering
Serving infrastructure
Metadata management
Process management
Monitoring

Monolithic batch pipeline architecture



Monolithic batch pipeline architecture

Features are not reusable (by your system or others)

If the data increases, you have to refactor the whole code to support PySpark or Ray

It’s hard to rewrite the prediction module in a more efficient language such as C++, Java, or Rust

It’s hard to share the work between multiple teams between the features, training, and prediction modules

It’s impossible to switch to streaming technology for real-time training

Google Cloud

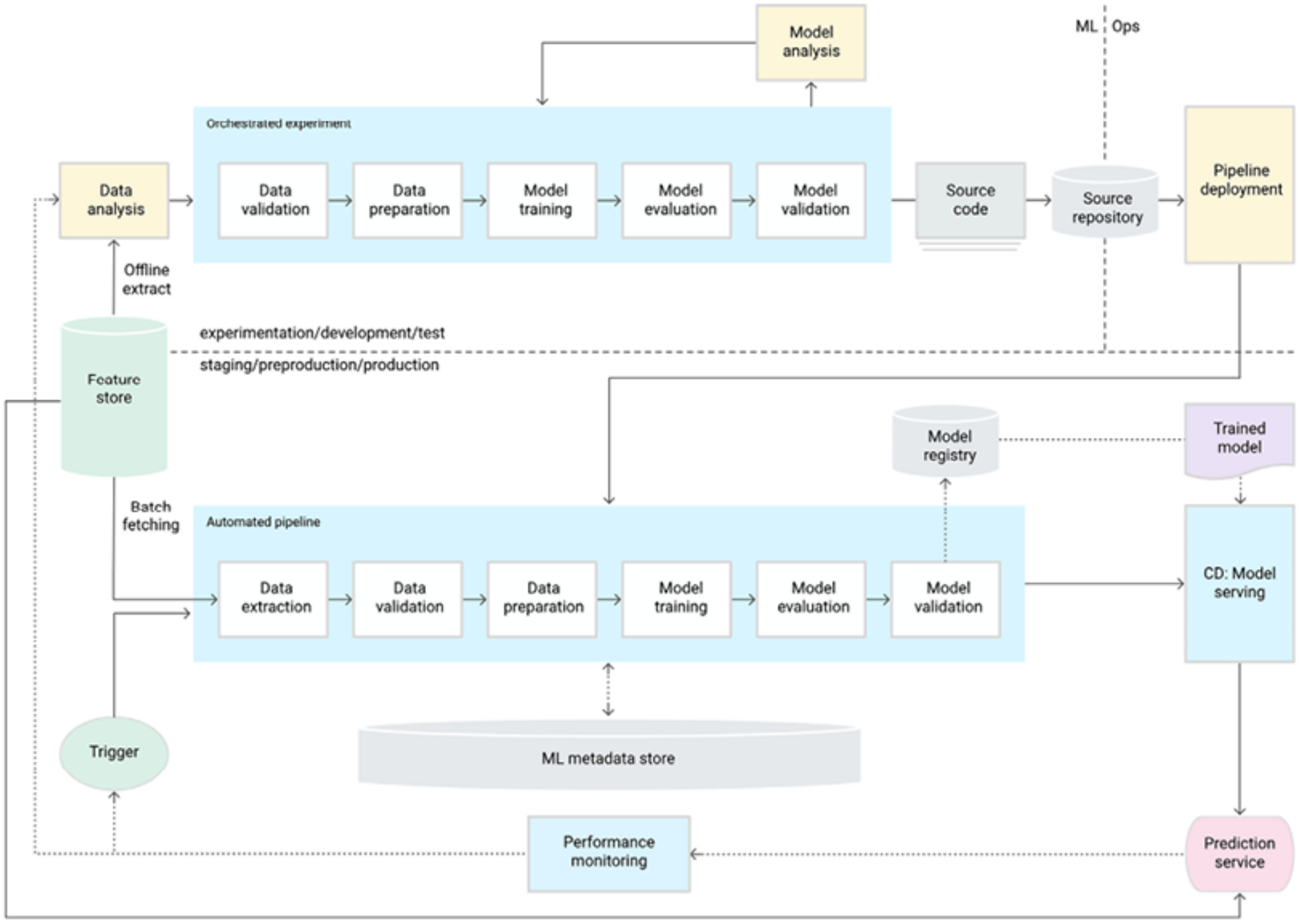
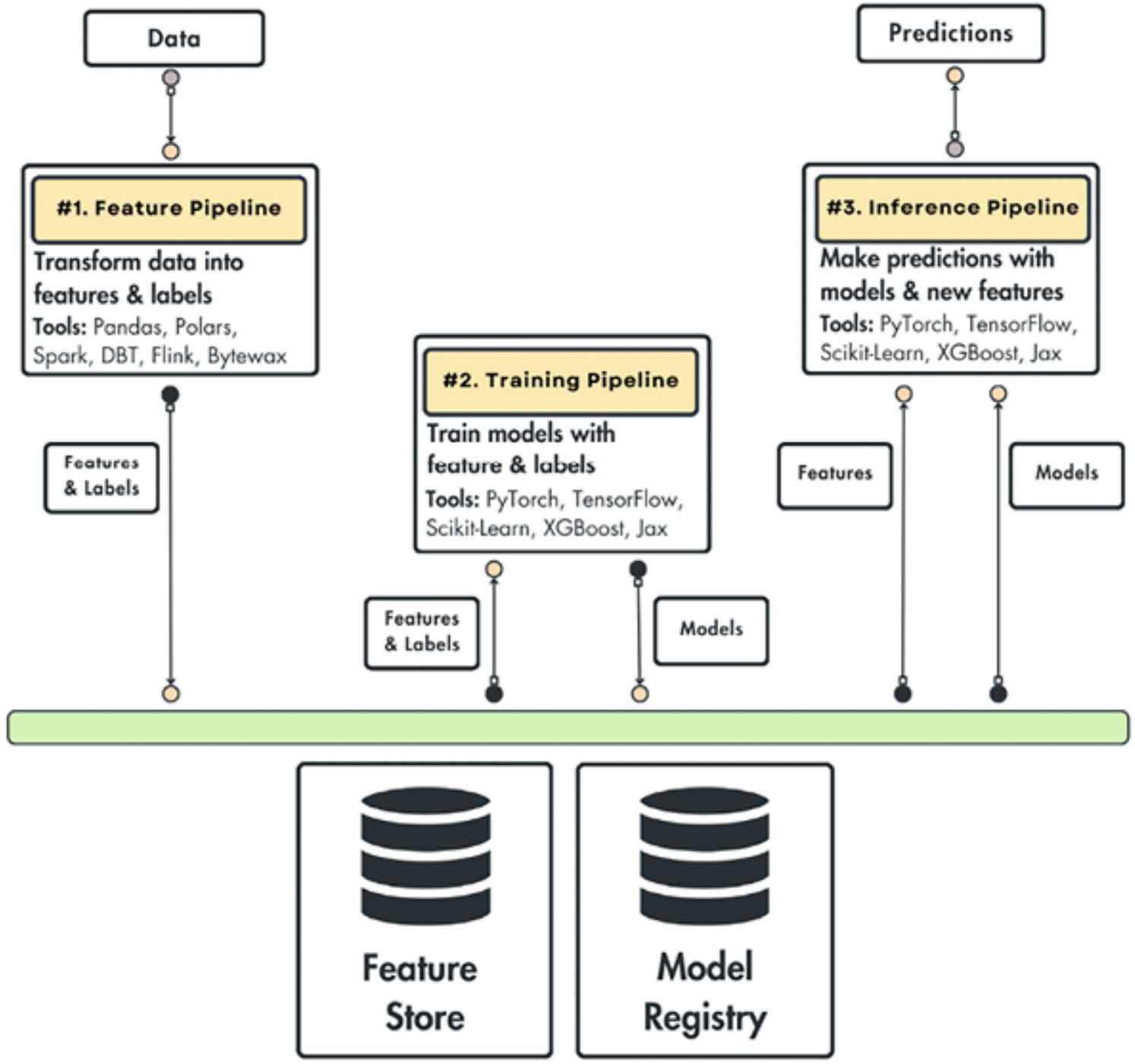
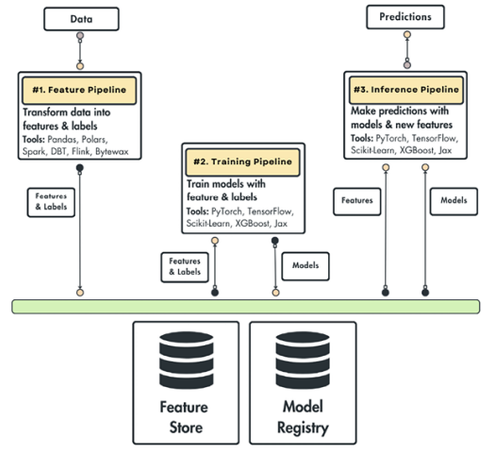


Figure 1.4: ML pipeline automation for CT (source: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning)

Feature, training, and inference (FTI) pipelines



Feature, training, and inference pipelines



- 3 independent components
- Each can have its own tech stack
- Each can be deployed intependently

Data Side

- Collect data autonomously and on a schedule
- Standardize the crawled data and store it in a data warehouse
- Clean the raw data
- Create instruct datasets for fine-tuning an LLM
- Chunk and embed the cleaned data.
- Store the vectorized data into a vector DB for RAG

Training

- Fine-tune LLMs of various sizes
- Fine-tune on instruction datasets of multiple sizes
- Switch between LLM types (for example, between Mistral, Llama, and GPT)
- Track and compare experiments
- Test potential production LLM candidates before deploying them
- Automatically start the training when new instruction datasets are available.

Inference Code needs

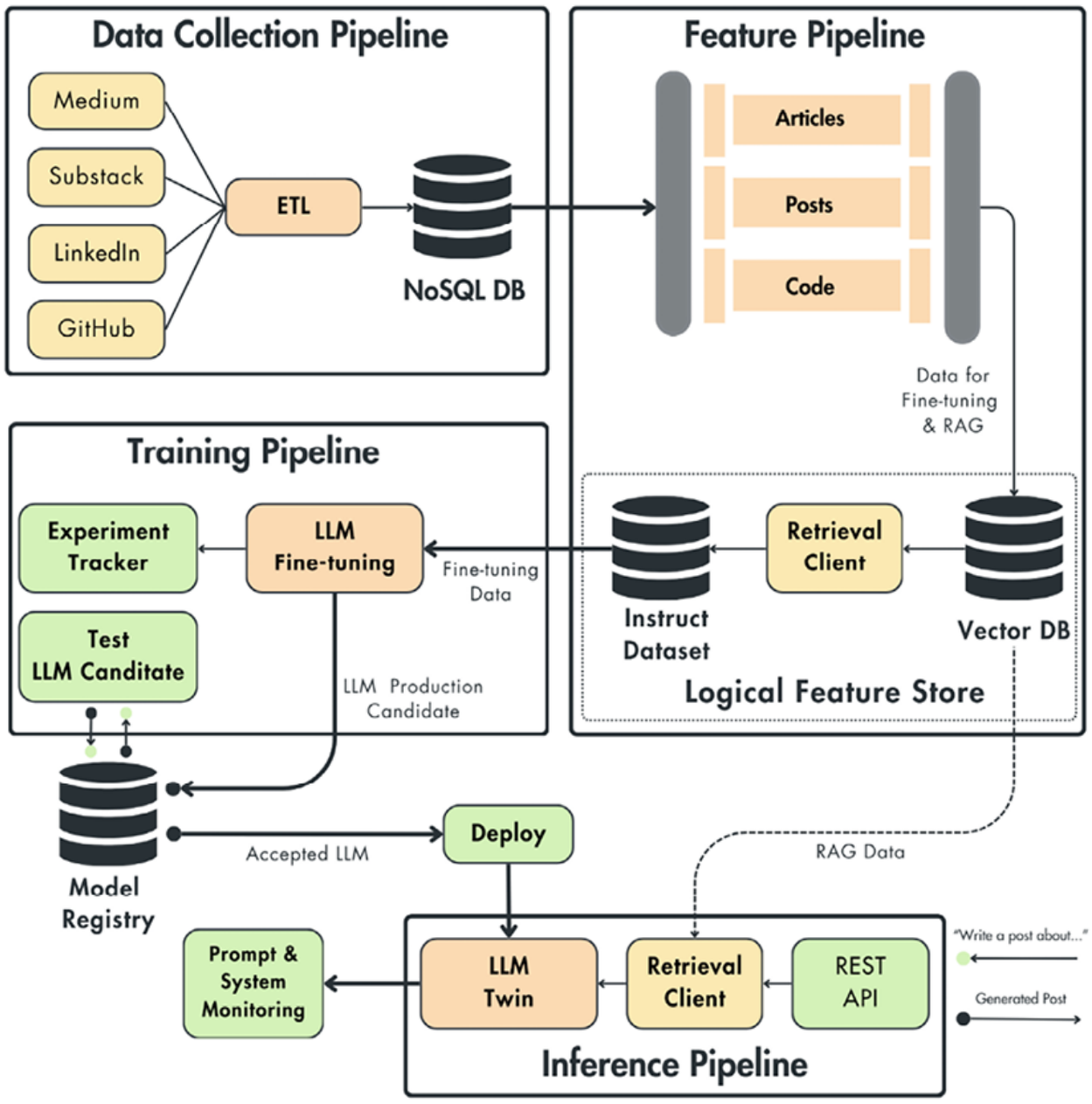
- A REST API interface for clients to interact with the LLM Twin
- Access to the vector DB in real time for RAG
- Inference with LLMs of various sizes
- Autoscaling based on user requests
- Automatically deploy the LLMs that pass the evaluation step.

The system needs to support

- Instruction dataset versioning, lineage, and reusability
- Model versioning, lineage, and reusability
- Experiment tracking
- Continuous training, continuous integration, and continuous delivery (CT/CI/CD)
- Prompt and system monitoring

FTI pipeline design

The data engineering team owns the data pipeline
The ML engineering team owns the FTI pipelines.



Feature Pipeline

Cleaning, chunking, and embedding
Two snapshots - fine-tuning, embedding

Training Pipeline

LLM agnostic pipeline?
What fine-tuning techniques to use?
How to scale the fine-tuning algorithm
How to pick an LLM production candidate?
How do you test the LLM?

Tooling

pyenv

poetry

Tooling - pyenv

Command-line tool to manage multiple Python versions

Multiple Python versions

Project-Specific versions

The correct version of Python will be used when in the project directory

pyinstall 3.11.8

Tooling - Poetry

Dependency and virtual environment management

Poetry resolves and installs project dependencies

pyproject.toml

Contains all the dependencies

```
[tool.poetry]
name = "llm-engineering"
version = "0.1.0"
description = ""
authors = ["iusztinpaul <p.b.iusztin@gmail.com>"]
license = "MIT"
readme = "README.md"

[tool.poetry.dependencies]
python = "~3.11"
zenml = { version = "0.74.0", extras = ["server"] }
pymongo = "^4.6.2"
click = "^8.0.1"
loguru = "^0.7.2"
rich = "^13.7.1"
numpy = "^1.26.4"
poethepoet = "0.29.0"
datasets = "^3.0.1"
```

```
poetry install --without aws
```

Tooling - Poe the Poet - task execution tool

Lightweight task runner

pyproject.toml

```
[tool.poe.tasks]
test = "pytest"
format = "black ."
start = "python main.py"
```

poetry poe test

poetry poe format

poetry poe start

Tooling - Docker

Lightweight container

Packages your application with only the libraries and dependencies it needs

Docker images can run on any machine that has Docker installed

.env

```
# OpenAI API Config
OPENAI_MODEL_ID=gpt-4o-mini
OPENAI_API_KEY=str

# Huggingface API Config
HUGGINGFACE_ACCESS_TOKEN=str

# Comet ML (during training and inference)
COMET_API_KEY=str

# --- Required settings when deploying the code. ---
# --- Otherwise, default values work fine. ---

# MongoDB database
DATABASE_HOST="mongodb://llm_engineering:llm_engineering@127.0.0.1:27017"

# Qdrant vector database
USE_QDRANT_CLOUD=false
QDRANT_CLOUD_URL=str
QDRANT_APIKEY=str

# AWS Authentication
AWS_ARN_ROLE=str
AWS_REGION=eu-central-1
AWS_ACCESS_KEY=str
AWS_SECRET_KEY=str
```

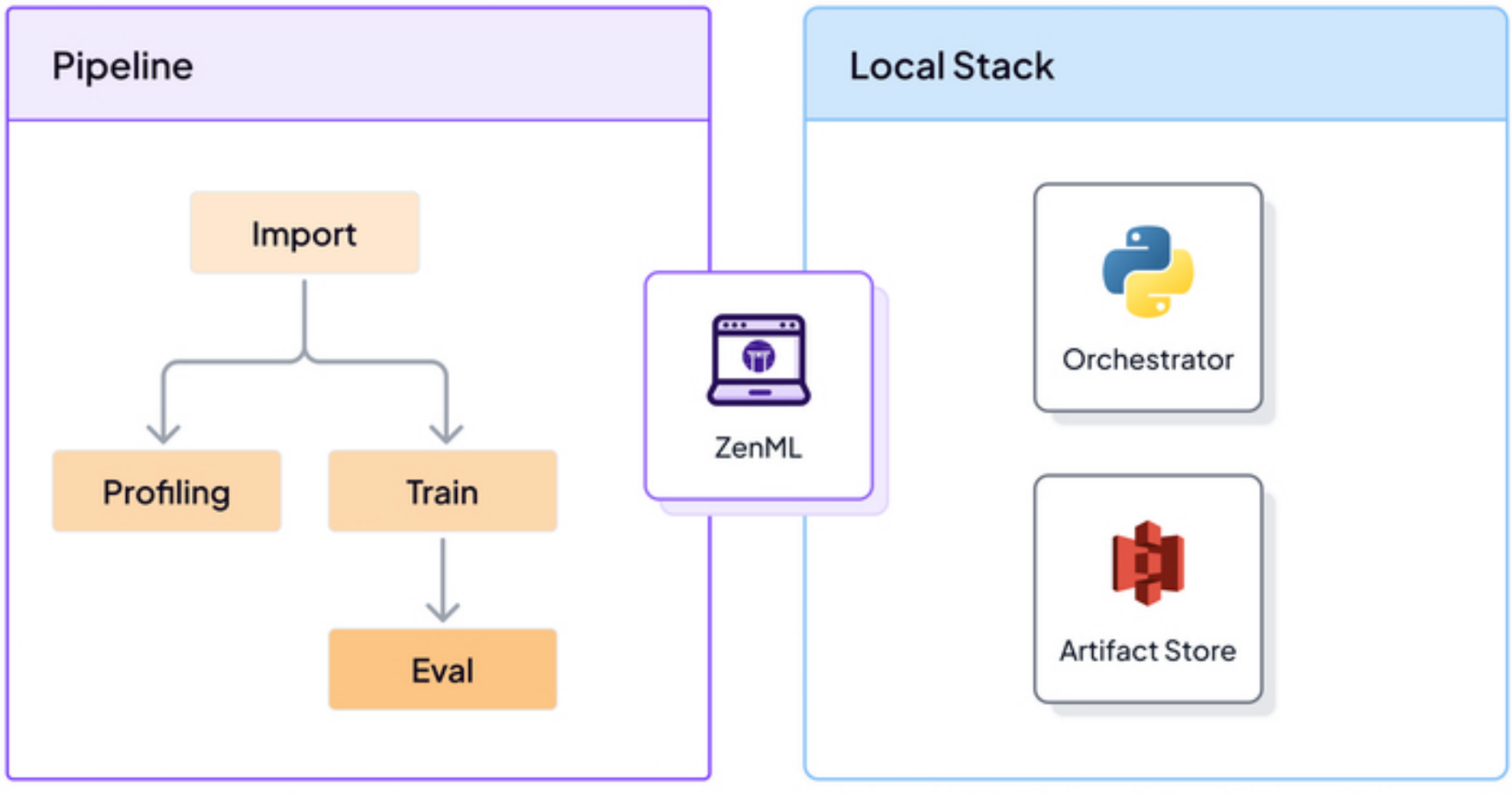
ZenML: orchestrator, artifacts, and metadata

Develop, orchestrate, and maintain reproducible machine learning pipelines

Orchestrator - coordinates all the steps to run in a pipeline

Artifact Store

Houses all data that passes through the pipeline as inputs and outputs



ZenML: orchestrator, artifacts, and metadata

Pipeline-Oriented Architecture

pipelines—step-by-step processes
Each step runs in its own environment

Reproducibility and Versioning

Tracks artifacts and metadata
Roll back and compare runs.

Works with tracking tools to record experiments and results

Can orchestrate cloud operations

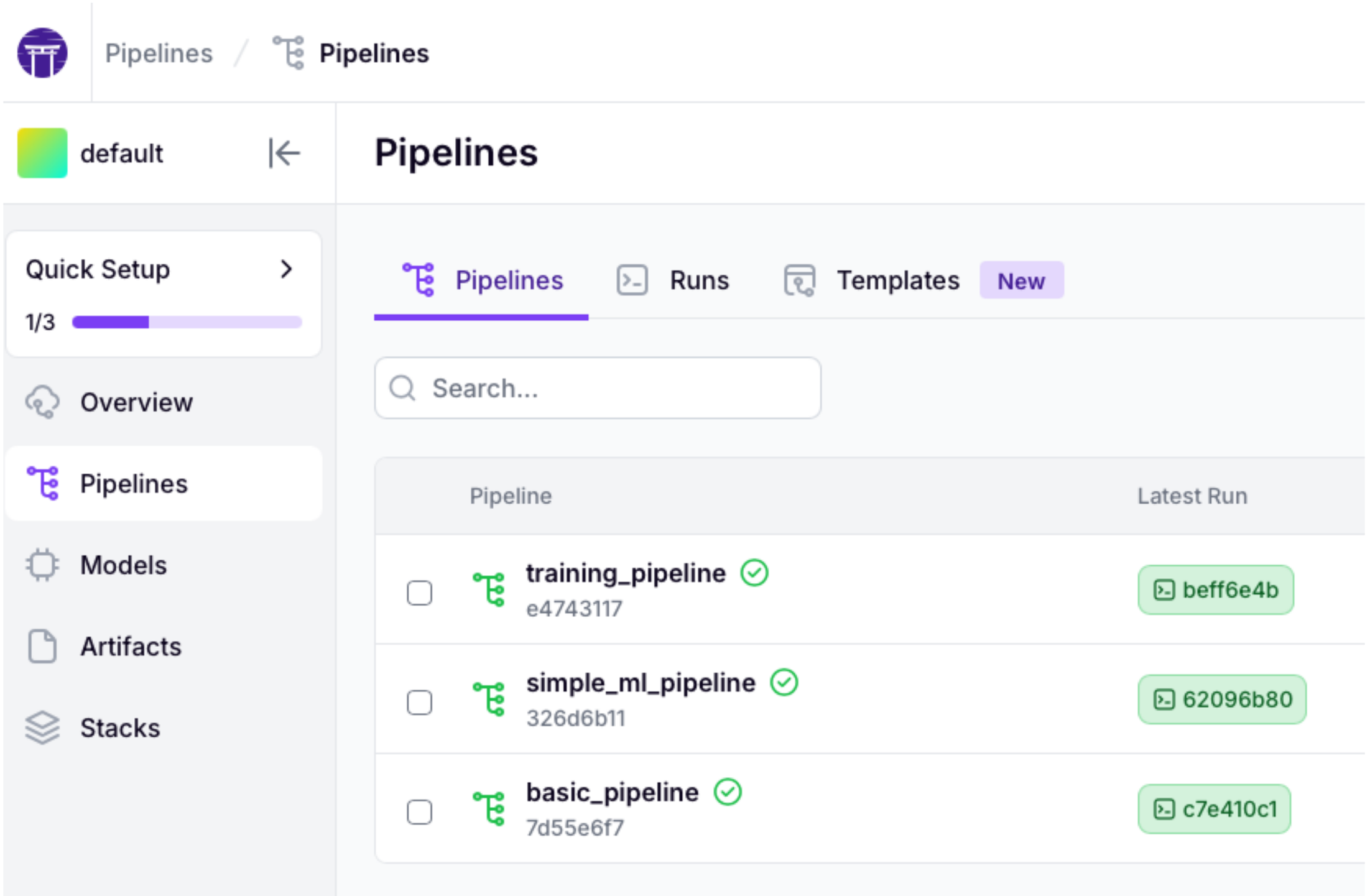
File: my.py

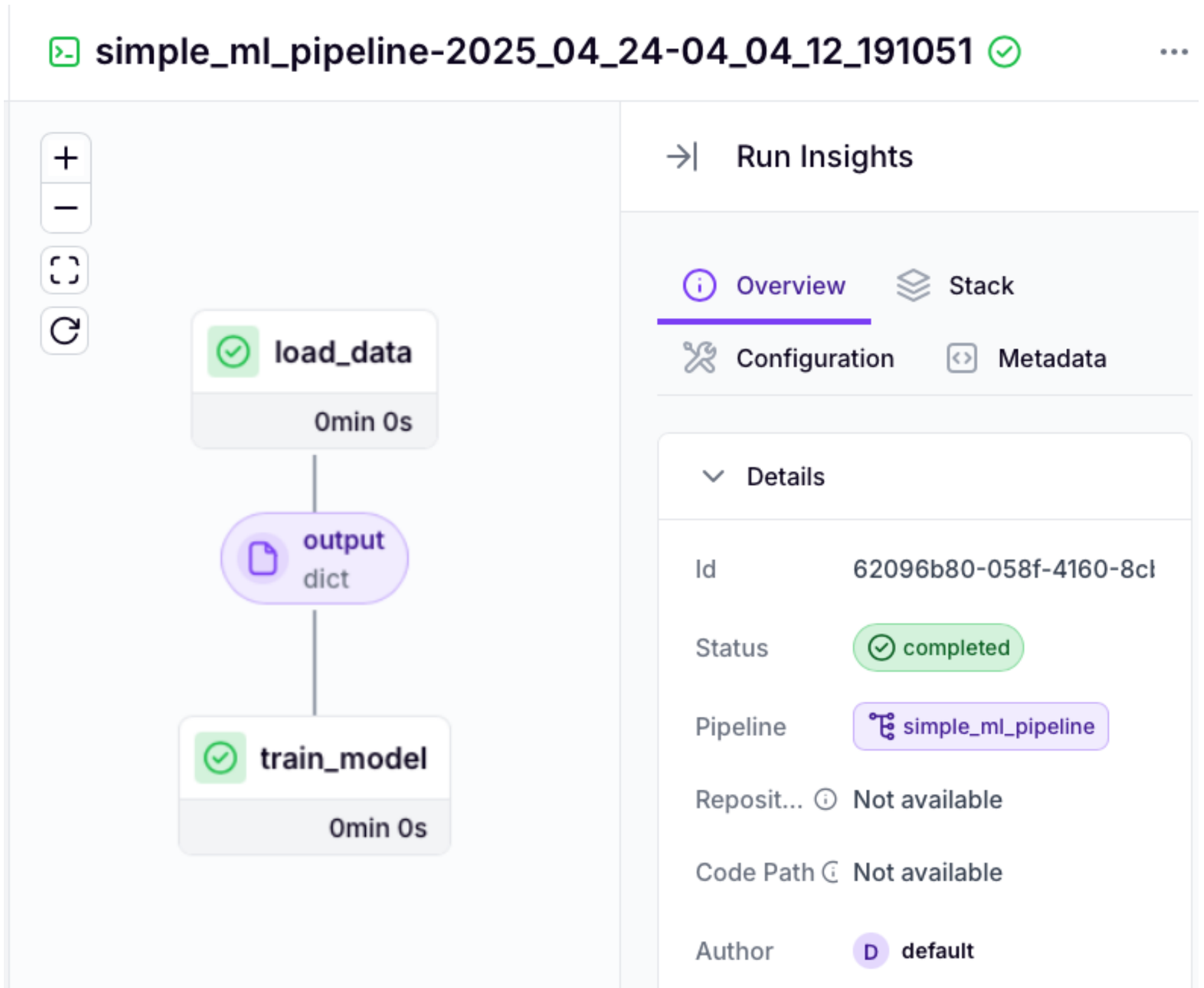
```
from zenml import pipeline, step

@step
def load_data() -> dict:
    training_data = [[1, 2], [3, 4], [5, 6]]
    labels = [0, 1, 0]
    return {'features': training_data, 'labels': labels}

@step
def train_model(data: dict) -> None:
    total_features = sum(map(sum, data['features']))
    total_labels = sum(data['labels'])

    print(f"Trained model using {len(data['features'])} data points. "
          f"Feature sum is {total_features}, label sum is {total_labels}")

@pipeline
def simple_ml_pipeline():
    dataset = load_data()
    train_model(dataset)

if __name__ == "__main__":
    run = simple_ml_pipeline()
```

rwhitney@127 test % python ml.py

```
Initiating a new run for the pipeline: simple_ml_pipeline.
Registered new pipeline: simple_ml_pipeline.
Using user: default
Using stack: default
  artifact_store: default
  orchestrator: default
Dashboard URL for Pipeline Run: http://127.0.0.1:8237/runs/62096b80-058f-4160-8cb2-9e2e1cabeb7
Step load_data has started.
Step load_data has finished in 0.068s.
Step train_model has started.
[train_model] Trained model using 3 data points. Feature sum is 21, label sum is 1
Step train_model has finished in 0.423s.
Pipeline run has finished in 0.600s.
```

Local Dashboard




simple_ml_pipeline-2025_04_24-04_04_12_191051
load_data
0min 0s
output
dict
train_model
0min 0s
Run Insights
Overview
Stack
Configuration
Metadata
Details
Id
62096b80-058f-4160-8cl
Status
completed
Pipeline
simple_ml_pipeline
Reposit...
Not available
Code Path
Not available
Author
D
default

Accessing

Data

URI

/Users/rwhitney/Library/Application Support/zenml/local_stores/d2b74feb-5a9f-4694…

Artifact Store default

Data Type dict

Code

```
from zenml.client import Client

artifact = Client().get_artifact_version("bb464528-a664-4221-84fd-d950df5265c3")
data = artifact.load()
```

Iris Example

```
from typing_extensions import Tuple, Annotated
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.base import ClassifierMixin
from sklearn.svm import SVC

from zenml import pipeline, step

@step
def training_data_loader() -> Tuple[
    Annotated[pd.DataFrame, "X_train"],
    Annotated[pd.DataFrame, "X_test"],
    Annotated[pd.Series, "y_train"],
    Annotated[pd.Series, "y_test"],
]:
    """Load the iris dataset as tuple of Pandas DataFrame / Series."""
    iris = load_iris(as_frame=True)
    X_train, X_test, y_train, y_test = train_test_split(
        iris.data, iris.target, test_size=0.2, shuffle=True, random_state=42
    )
    return X_train, X_test, y_train, y_test
```

Iris Example

```
@step
def svc_trainer(
    X_train: pd.DataFrame,
    y_train: pd.Series,
    gamma: float = 0.001,
) -> Tuple[
    Annotated[ClassifierMixin, "trained_model"],
    Annotated[float, "training_acc"],
]:
    """Train a sklearn SVC classifier and log to MLflow."""
    model = SVC(gamma=gamma)
    model.fit(X_train.to_numpy(), y_train.to_numpy())
    train_acc = model.score(X_train.to_numpy(), y_train.to_numpy())
    print(f"Train accuracy: {train_acc}")
    return model, train_acc


@pipeline
def training_pipeline(gamma: float = 0.002):
    X_train, X_test, y_train, y_test = training_data_loader()
    svc_trainer(gamma=gamma, X_train=X_train, y_train=y_train)
```

Iris Example

```
@pipeline
def training_pipeline(gamma: float = 0.002):
    X_train, X_test, y_train, y_test = training_data_loader()
    svc_trainer(gamma=gamma, X_train=X_train, y_train=y_train)


if __name__ == "__main__":
    training_pipeline()
```

Hyper-Parameter Tuning

```
from typing import Annotated
from sklearn.base import ClassifierMixin
from zenml import step

MODEL_OUTPUT = "model"

@step
def train_step(learning_rate: float) -> Annotated[ClassifierMixin, MODEL_OUTPUT]:
"""Train a model with the given learning-rate."""
# <your training code goes here>
...
```

Hyper-Parameter Tuning

```
from zenml import pipeline
from zenml import get_step_context, step
from zenml.client import Client

@step
def selection_step(step_prefix: str, output_name: str):
    """Pick the best model among all training steps."""
    run = Client().get_pipeline_run(get_step_context().pipeline_run.name)
    trained_models = {}
    for step_name, step_info in run.steps.items():
        if step_name.startswith(step_prefix):
            model = step_info.outputs[output_name][0].load()
            lr = step_info.config.parameters["learning_rate"]
            trained_models[lr] = model

    # <evaluate and select your favorite model here>

@pipeline
def hp_tuning_pipeline(step_count: int = 4):
    after = []
    for i in range(step_count):
        train_step(learning_rate=i * 0.0001, id=f"train_step_{i}")
        after.append(f"train_step_{i}")

    selection_step(step_prefix="train_step_", output_name=MODEL_OUTPUT, after=after)
```

Hyper-Parameter Tuning

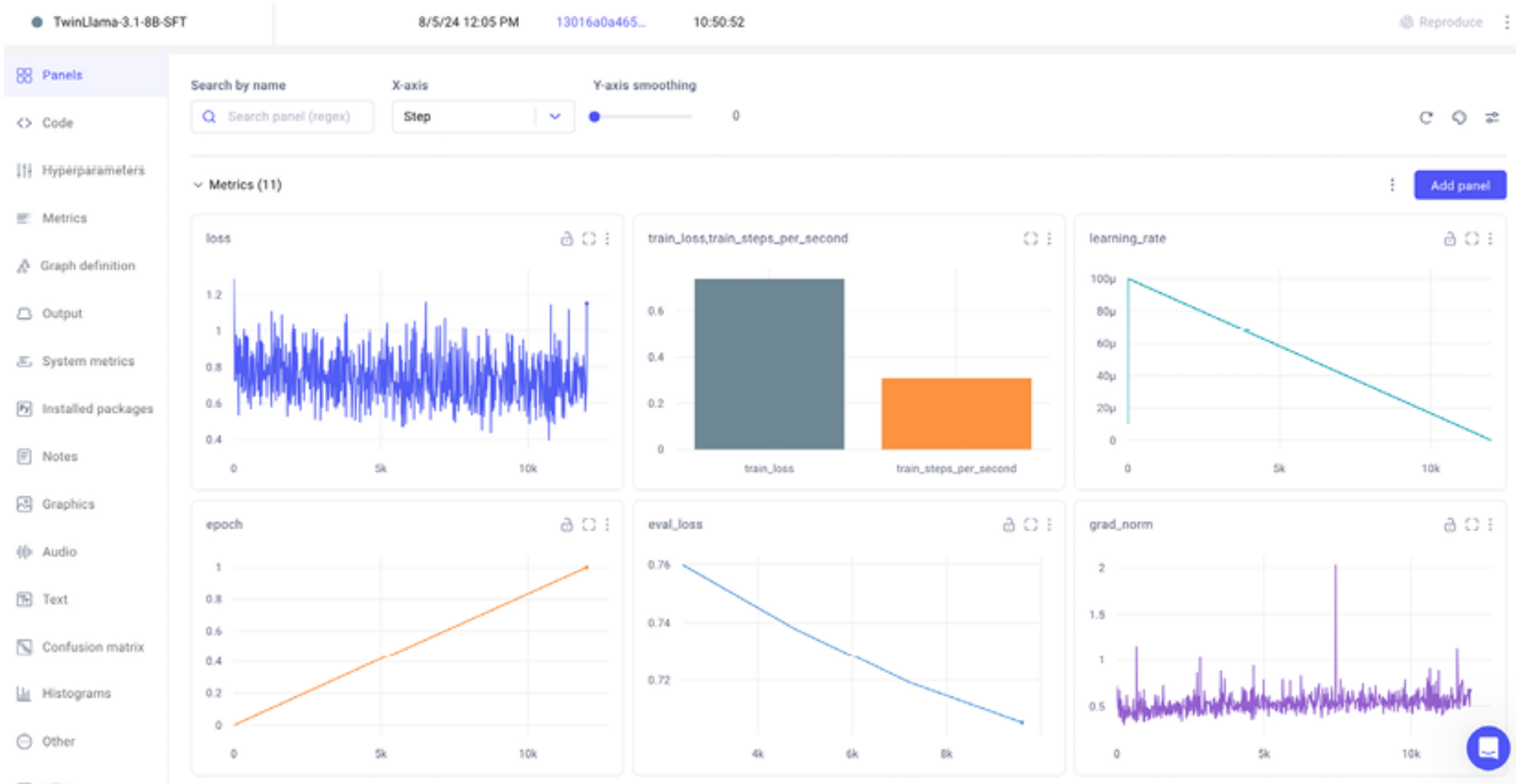
```
if __name__ == "__main__":
    hp_tuning_pipeline(step_count=4)()
```

Comet ML: experiment tracker

ML training is experimental and iterative

Comet ML logs

- metrics
- hyperparameters
- output files
- system configurations


TwinLlama-3.1-8B-SFT
8/5/24 12:05 PM
13016a0a465...
10:50:52
Reproduce
Panels
Code
Hyperparameters
Metrics
Graph definition
Output
System metrics
Installed packages
Notes
Graphics
Audio
Text
Confusion matrix
Histograms
Other
Search by name
Search panel (regex)
X-axis
Step
Y-axis smoothing
0
Metrics (11)
Add panel
loss
train_loss,train_steps_per_second
learning_rate
train_loss
train_steps_per_second
epoch
eval_loss
grad_norm

Opik

Open-source platform for evaluating, testing, and monitoring LLM applications

Development

- Track all LLM calls and traces
- Try out different prompts and models

Evaluation

- Store test cases and run experiments
- Use Opik's LLM as a judge metric

Production Monitoring

- Log production traces
- Dashboards
- Online evaluation metrics

Opik Logging

```
from opik.integrations.openai import track_openai
from openai import OpenAI

# Wrap your OpenAI client
openai_client = OpenAI()
openai_client = track_openai(openai_client)
```

```
from opik import track
import anthropic

@track
def call_llm(client, messages):
    return client.messages.create(messages=messages)

client = anthropic.Anthropic()

call_llm(client, [{"role": "user", "content": "Why is tracking and evaluation of LLMs important?"}])
```

Opik Logging

```
from langchain_openai import OpenAI
from langchain.prompts import PromptTemplate
from opik.integrations.langchain import OpikTracer

# Initialize the tracer
opik_tracer = OpikTracer()

# Create the LLM Chain using LangChain
llm = OpenAI(temperature=0)

prompt_template = PromptTemplate(
    input_variables=["input"],
    template="Translate the following text to French: {input}"
)

# Use pipe operator to create LLM chain
llm_chain = prompt_template | llm

# Generate the translations
llm_chain.invoke({"input": "Hello, how are you?"}, callbacks=[opik_tracer])
```

Prompts

```
import opik

# Create a new Prompt instance
prompt = opik.Prompt(
  name="Q&A Prompt",
  prompt="Hello, {{name}}! Welcome to {{location}}. How can I assist you today?",
  metadata={"temperature": 0.4}
)

# Format the prompt with the given parameters
formatted_prompt = prompt.format(name="Alice", location="Wonderland")
print(formatted_prompt)
```

Downloading the Prompt

```
import opik

client = opik.Opik()

# Get the most recent version of a prompt
prompt = client.get_prompt(name="Q&A Prompt")

# Read metadata from the most recent version of a prompt
print(prompt.metadata)

# Format the prompt with the given parameters
formatted_prompt = prompt.format(name="Alice", location="Wonderland")
print(formatted_prompt)
```

Databases

MongoDB: NoSQL database

Qdrant: vector database

AWS

SageMaker: training and inference compute