CS 668 Applied Large Language Models
Spring Semester, 2025
Doc 23 Vibe Programming, ML- & LLM-Ops
Apr 16, 2026

Stages of AI-Assisted Coding

Stage 1: Zero or Near-Zero AI

Maybe code completions,sometimes ask Chat questions

Stage 2: Coding agent in IDE, permissions turned on.

A narrow coding agent in a sidebar asks your permission to run tools.

Stage 3: Agent in IDE, YOLO mode

Trust goes up. You turn off permissions, agent gets wider.

Stage 4: In IDE, wide agent

Your agent gradually grows to fill the screen. Code is just for diffs.

Stage 5: CLI, single agent. YOLO.

Diffs scroll by. You may or may not look at them.

Stage 6: CLI, multi-agent, YOLO.

You regularly use 3 to 5 parallel instances. You are very fast.

Stage 7: 10+ agents, hand-managed.

You are starting to push the limits of hand-management.

Stage 8: Building your own orchestrator.

You are on the frontier, automating your workflow.

Welcome to Gas Town https://steve-yegge.medium.com/welcome-to-gas-town-4f25ee16dd04

13 Local AI Coding Models

Small Hardware (8–16GB RAM)

- Phi-3 Mini
- gpt-micro
- Qwen2.5-Coder-7B

Mid-Range GPU (16–32GB VRAM)

- StarCoder2–15B
- CodeLlama-13B
- Granite-Code-20B
- Qwen2.5-Coder-14B

Mid-Range GPU (16–32GB VRAM)

- StarCoder2–15B
- CodeLlama-13B
- Granite-Code-20B
- Qwen2.5-Coder-14B

https://medium.com/@shouke.wei/13-local-ai-coding-models-you-should-know-in-2025-22934f202873

The Golden Rules of AI Coding

Be specific and clear about what you want

Always validate AI output against your intent

Treat AI as a junior developer (with supervision)

Use AI to expand your capabilities, not replace your thinking

Coordinate upfront among the team before generating code

Treat AI usage as a normal part of the development conversation

Isolate AI changes in Git by doing separate commits

Ensure that all code, whether human or AI-written, undergoes code review

Don’t merge code you don’t understand

Prioritize documentation, comments, and ADRs

Share and reuse effective prompts

Regularly reflect and iterate

Beyond Vibe Coding, **Andy Osmani**, O’Reilly Media, August 2025

Be specific and clear about what you want

Write a sorting function

Write a Python function sort_by_lastname(customers) that takes a list of customer records (each with a first_name and last_name field) and returns a list sorted by last_name alphabetically. Include a brief docstring and handle the case of missing last names by treating them as empty strings.

Be specific and clear about what you want

Mention the language or environment

If you want a solution in JavaScript, say so: “Write a JavaScript function...” versus just “Write a function...”

If you want it for a specific framework or version, include that (“Using React Hooks...” or “in Python 3...”).

Define the scope of the output

Do you want just a single function? A full file or module? Tests included?

For example, “Provide only the function implementation” and “Provide a complete runnable script” can yield different responses.

Be specific and clear about what you want

Include requirements and constraints

Think of edge cases or constraints and put them in the prompt.

If you need the code to be optimized for performance or use a certain algorithm, say so: “using O(n) time and O(1) space” or “using a binary search approach.”

Avoid ambiguous references

Don’t use words like it without a clear antecedent.

Instead of “Process it and return the result,” say, “Process the array and return the resulting array.”

Be specific and clear about what you want

Name your desired output format

If you want the AI to output just code or code with comments or an explanation, you can instruct that:

“Give only the code, no explanation” or

“Provide code and a brief comment for each step.”

Be specific and clear about what you want

Here’s what not to do:

Don’t write a whole novel

Be concise but complete in your description.

For instance, you usually don’t need to preface with “You are a world-class programmer...” in a coding context

Don’t assume the AI will fill in details by itself correctly

If something is important (like thread safety), mention it.

If it’s not mentioned, assume the AI might not handle it.

Avoid open-ended “creative” prompts when you need deterministic outputs

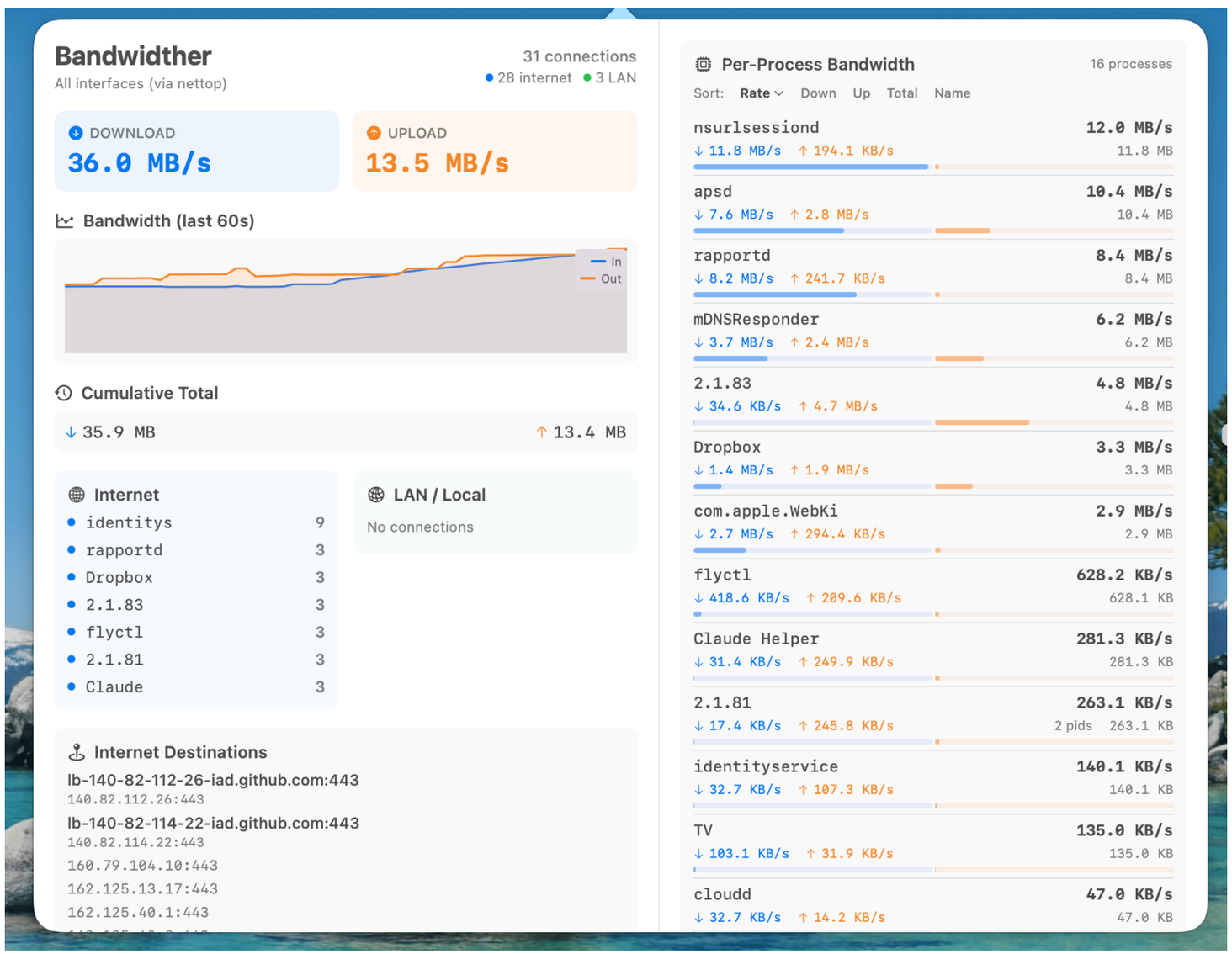
For example, saying, “Write some code to analyze data” might cause the AI to guess what analysis you want. Instead, specify:

Vibe coding SwiftUI apps is a lot of fun

https://simonwillison.net/2026/Mar/27/vibe-coding-swiftui/

Simon Willison, Mar 27, 2026

Bandwidther shows what apps are using network bandwidth



Vibe coding SwiftUI apps is a lot of fun

Claude Code transcript

https://gisthost.github.io/?6e06d4724c64c10d1fc3fbe19d9c8575/index.html

25 prompts · 246 messages · 80 tool calls · 4 commits · 5 pages

Show me how much network bandwidth is in use from this machine to the internet as opposed to local LAN

mkdir /tmp/bandwidther and write a native Swift UI app in there that shows me these details on a live ongoing basis

git init and git commit what you have so far

Now suggest features we could add to that app, the goal is to provide as much detail as possible concerning network usage including by different apps

add Per-Process Bandwidth, relaunch the app once that is done

I think it crashed

[Image #1] I do not understand why it says 381KB/s total download but then 24 MB/s for Dropbox - how can that second number be highecr than the first?

```
import SwiftUI

struct SparklineView: View {
    let data: [Double]
    let color: Color
    let maxValue: Double

    var body: some View {
        GeometryReader { geo -> AnyView in
            let w = geo.size.width
            let h = geo.size.height
            let safeMax = maxValue > 0 ? maxValue : 1
let count = data.count

let points: [CGPoint] = data.enumerated().map { i, val in
    let x = count > 1 ? w * Double(i) / Double(count - 1) : 0
    let y = h - min(max(val / safeMax * h, 0), h)
    return CGPoint(x: x, y: y)
}

let fill = Path { path in
    guard count > 1 else { return }
    path.move(to: CGPoint(x: 0, y: h))
    points.forEach { path.addLine(to: $0) }
```

```
import Foundation
import Darwin

struct ConnectionGroup {
    var count: Int = 0
    var ips: [String: Int] = [:]
}

struct ProcessStat: Identifiable {
    let id: String
    let name: String
    let bytesIn: Double   // bytes/sec
    let bytesOut: Double  // bytes/sec
}

class NetworkMonitor: ObservableObject {
    @Published var downloadRate: Double = 0
    @Published var uploadRate: Double = 0
    @Published var downloadHistory: [Double] = Array(repeating: 0, count: 60)
    @Published var uploadHistory: [Double] = Array(repeating: 0, count: 60)
    @Published var lanGroup = ConnectionGroup()
    @Published var internetGroup = ConnectionGroup()
    @Published var topProcesses: [ProcessStat] = []
    @Published var availableInterfaces: [String] = []
```

MLOps & LLMOps

MLOps

Classic machine learning pipeline

Evaluation is objective.

Clear metrics: Accuracy, F1-score,R^2, or Mean Squared Error (MSE)

LLMOps

Unique headaches of non-deterministic, massive, and expensive generative models

Evaluation is highly subjective. LLMs can "hallucinate" while maintaining perfect grammar.

MLOps & LLMOps

Feature	MLOps	LLMOps
Primary Goal	Prediction / Classification	Generation / Reasoning
Data Focus	Feature Engineering	Prompt Engineering & RAG
Cost Center	Data Labeling	Compute (Training & Inference)
Feedback Loop	Statistical Drifts	Human Feedback (RLHF/DPO)
Main Tools	Scikit-learn, MLflow, Airflow	Hugging Face, LangChain, Vector DBs

Scale Changes Everything

651 ML & LLMOps

Reproducibility and Experiment Tracking: MLflow or Weights & Biases, DVC basics, reproducible experiments

DVC (Data Version Control) is a tool that:

- Tracks datasets, models, and intermediate outputs
- Defines pipelines (like training workflows)
- Stores large files outside Git (S3, GCS, Azure, local, etc.)
- Keeps lightweight pointers in Git so everything stays versioned together

ML systems break when:

- Data changes silently
- Models aren’t reproducible
- Pipelines aren’t tracked

651 ML & LLMOps

Data and Model Versioning
DVC pipelines,
ML repo structuring,
Git+DVC integration

```
dvc stage add -n train \
  -d src/train.py \
  -d data/processed/train.csv \
  -d params.yaml \
  -o models/model.pkl \
  python src/train.py
```

651 ML & LLMOps

Data and Model Versioning
DVC pipelines,
ML repo structuring,
Git+DVC integration

```
dvc stage add -n train \
  -d src/train.py \
  -d data/processed/train.csv \
  -d params.yaml \
  -o models/model.pkl \
  python src/train.py
```

Model Development and Evaluation:
Scikit-learn baseline model,
metrics for churn,
bias checks

Model Serving and APIs:
Serving models with FastAPI or Flask,
RESTful inference endpoints

Monitoring and Logging:
Logging with Prometheus and Grafana,
drift detection,
prediction logs

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

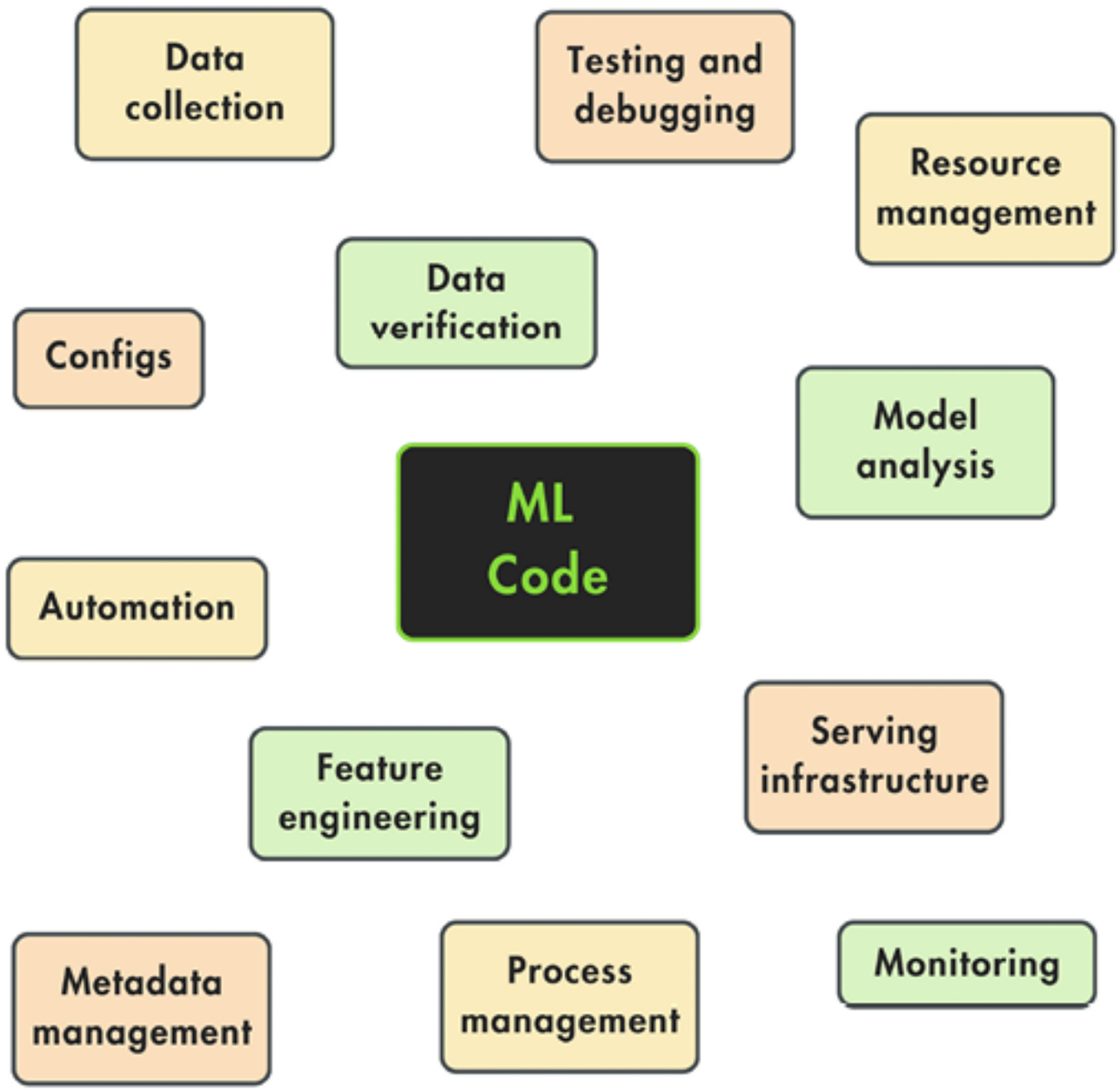
Version, track, and share the datasets and models

Monitor your infrastructure and models

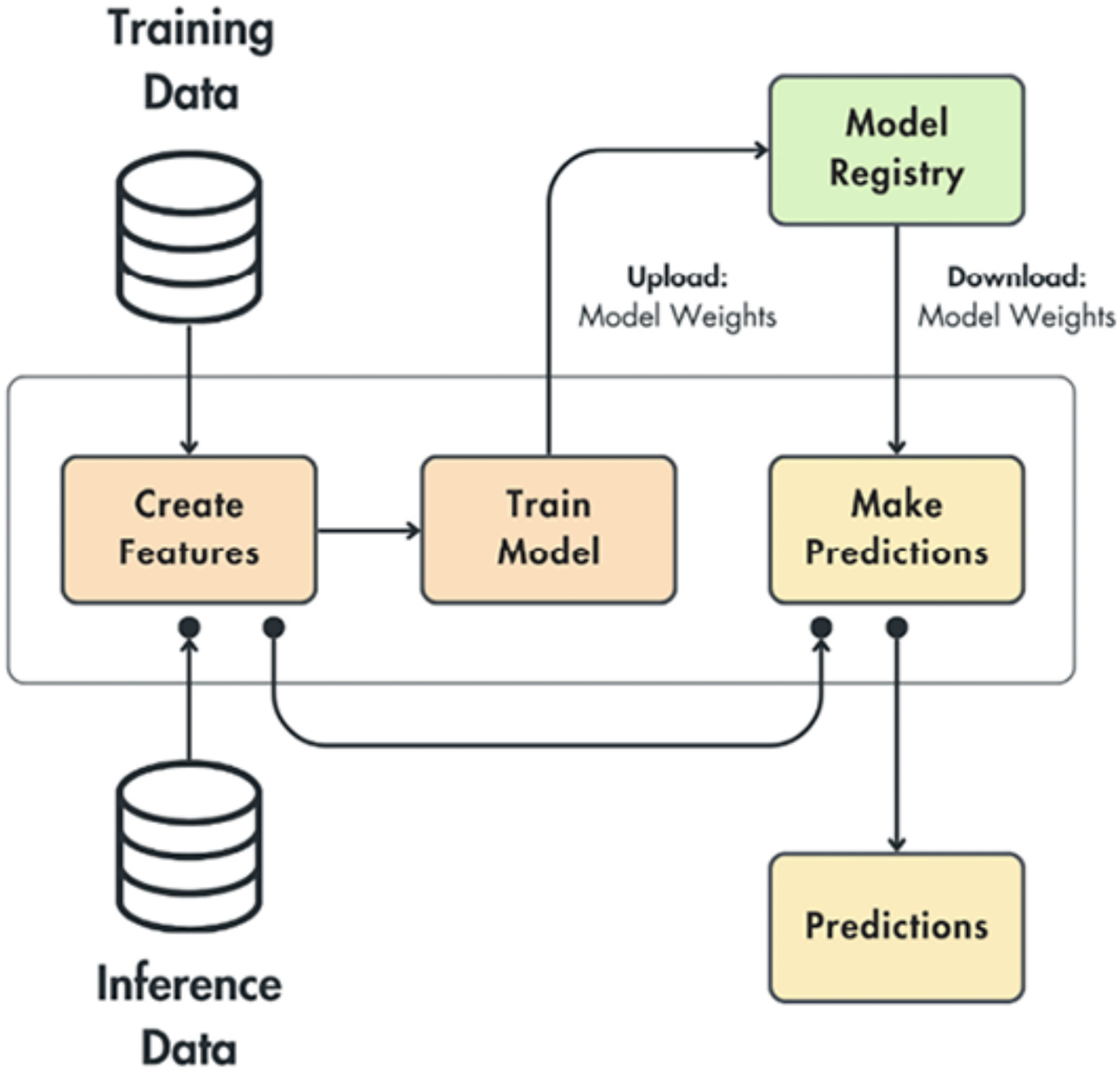
Deploy the model on a scalable infrastructure

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)

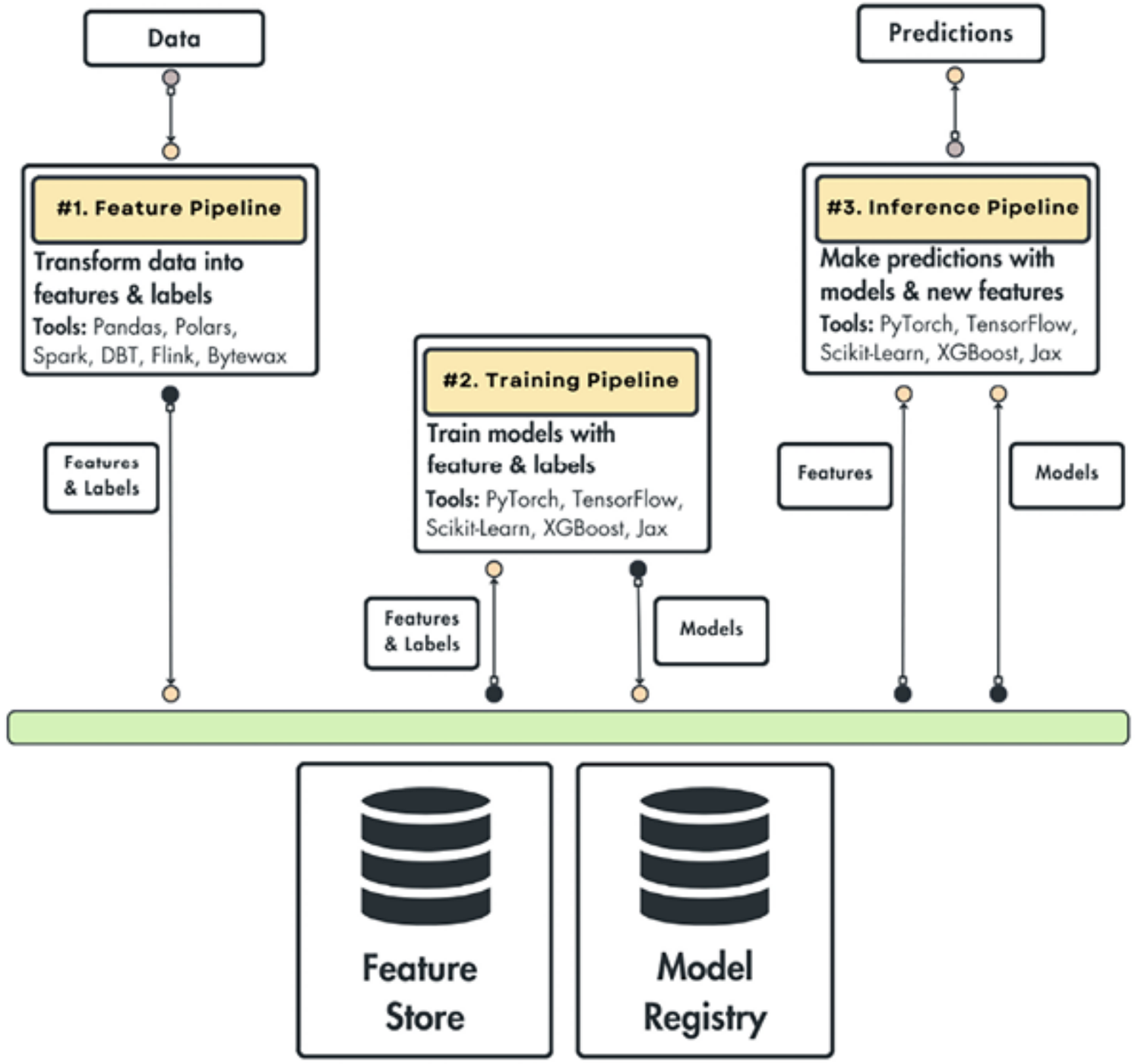
If the data increases, you have to refactor the whole code to support PySpark or Ray

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

Feature, training, and inference (FTI) pipelines



Google Cloud

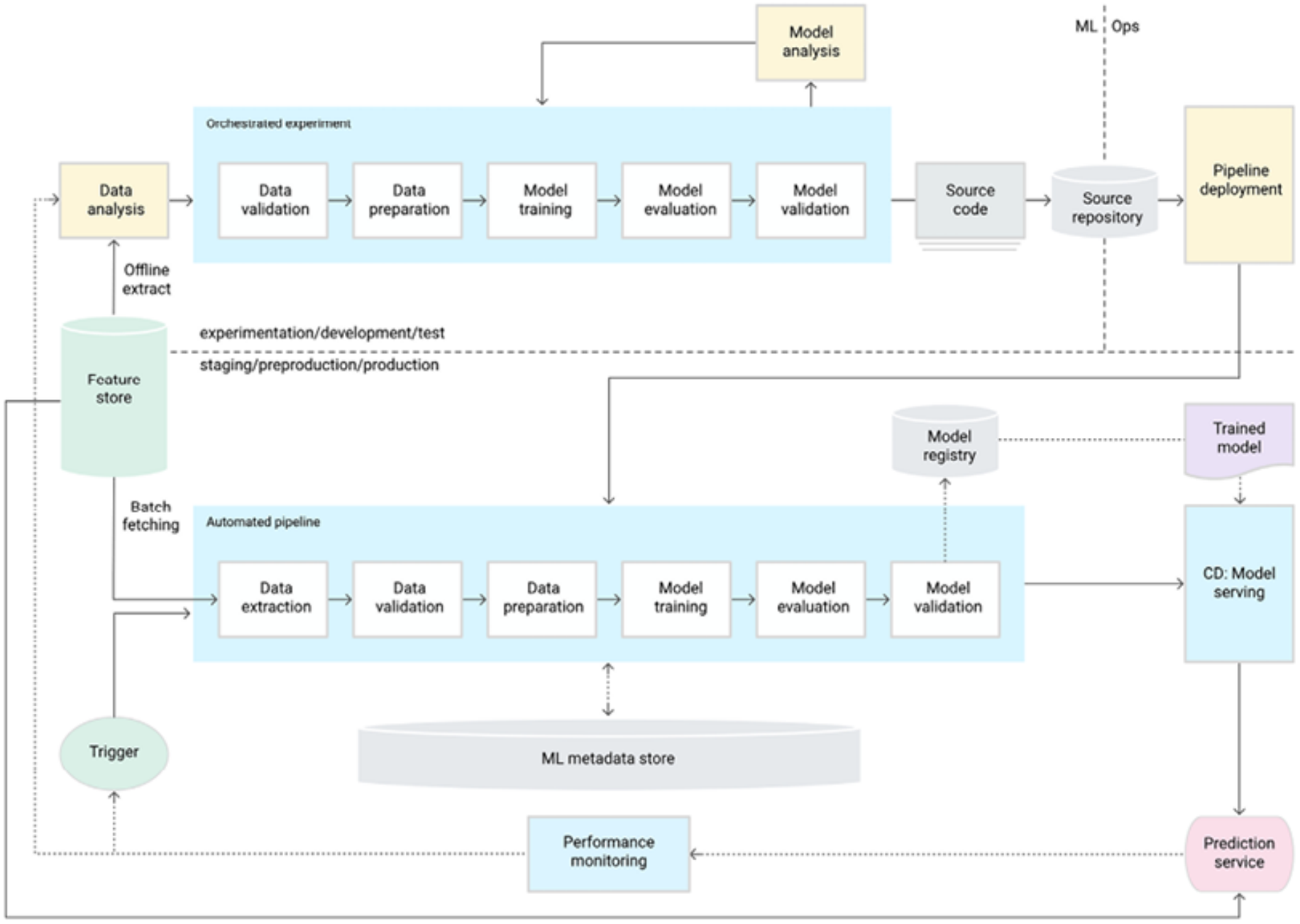
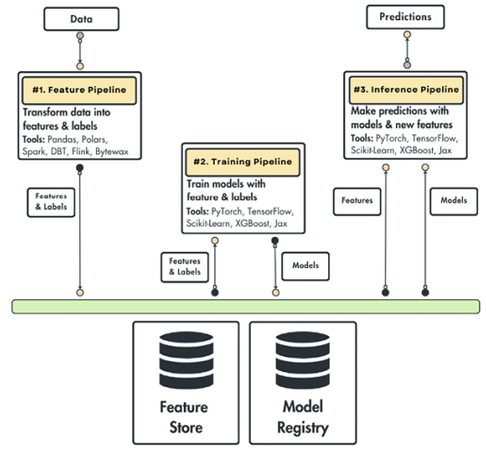


Figure 1.4: ML pipeline automation for CT (source: https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning)

Feature, training, and inference pipelines



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

Data

- 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

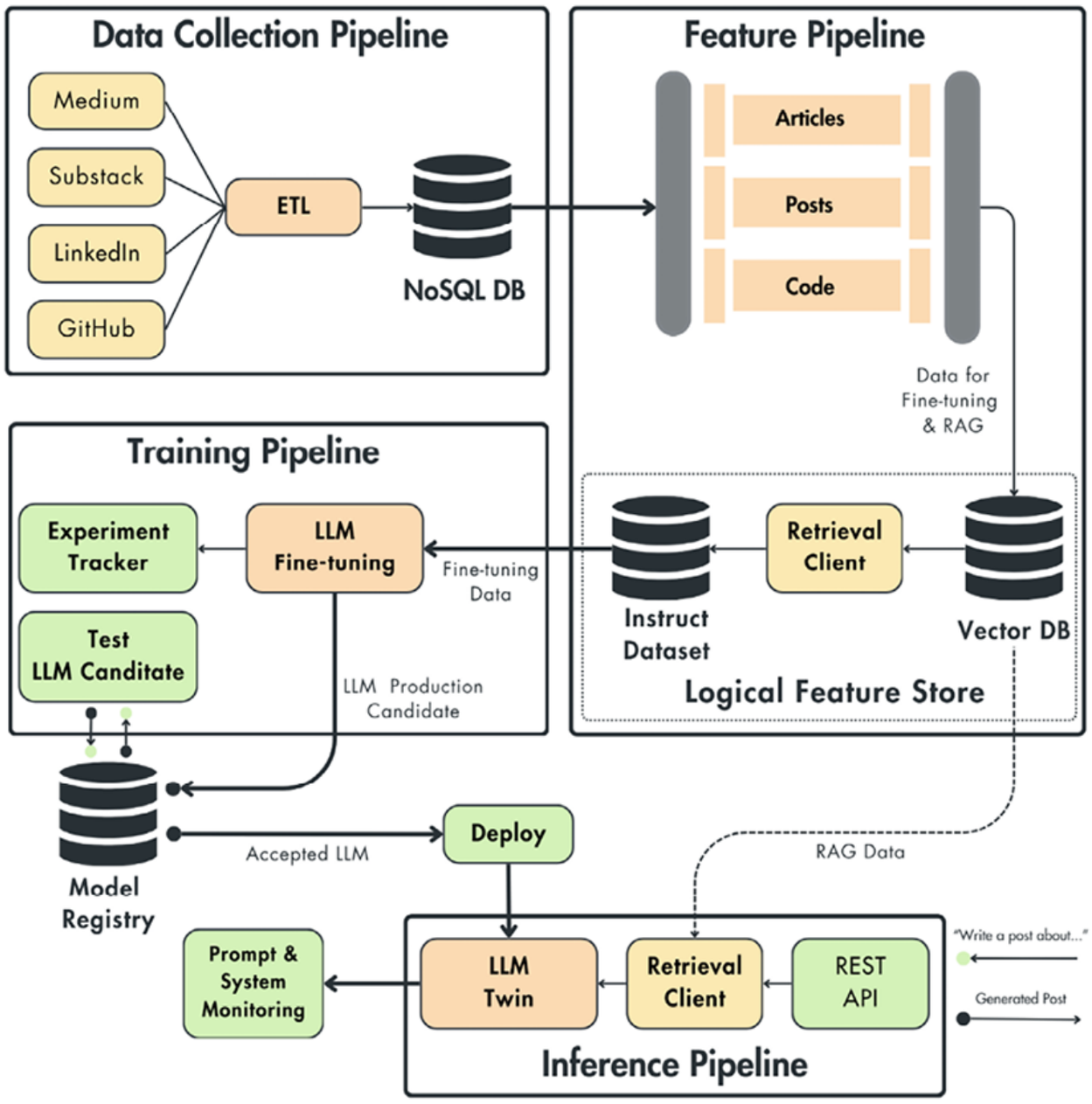
- A REST API interface for clients to interact with the LLM Twin
- Access to the vector DB in real time for RAG
- Inference with LLMs of various sizes
- Autoscaling based on user requests
- Automatically deploy the LLMs that pass the evaluation step.

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

Uv

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

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

```
[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"
```

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

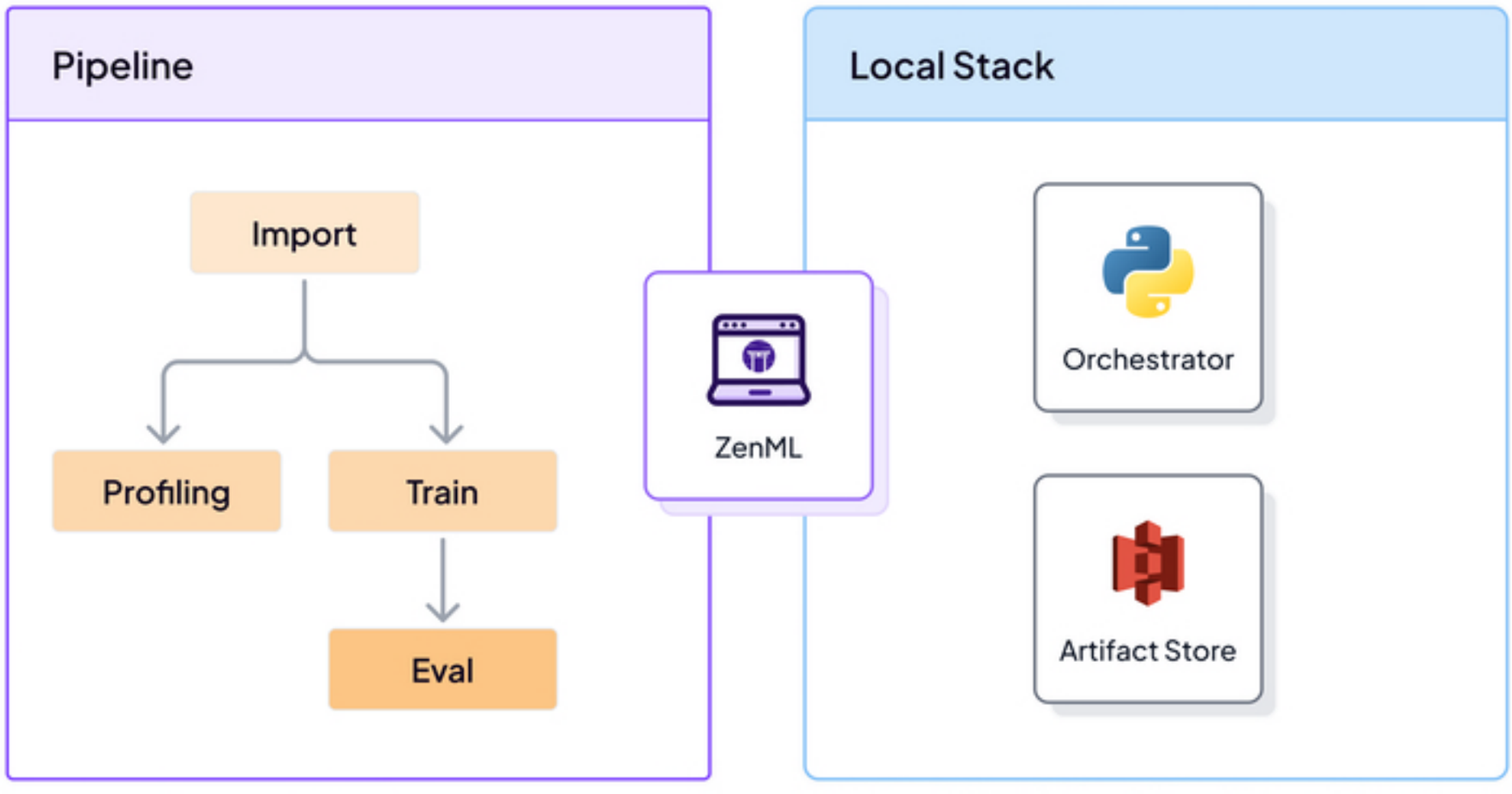
ZenML: orchestrator, artifacts, and metadata

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

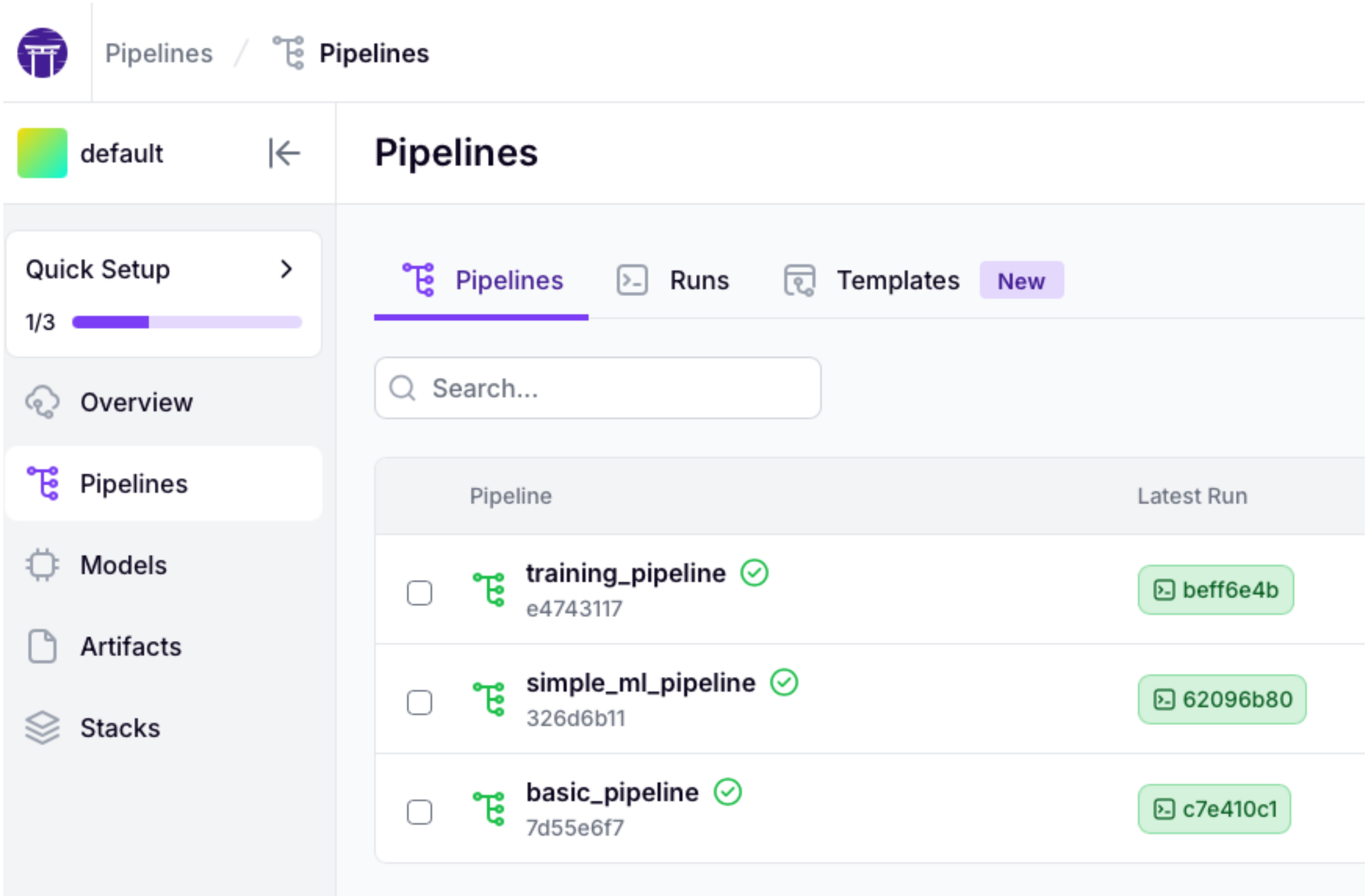
File: my.py

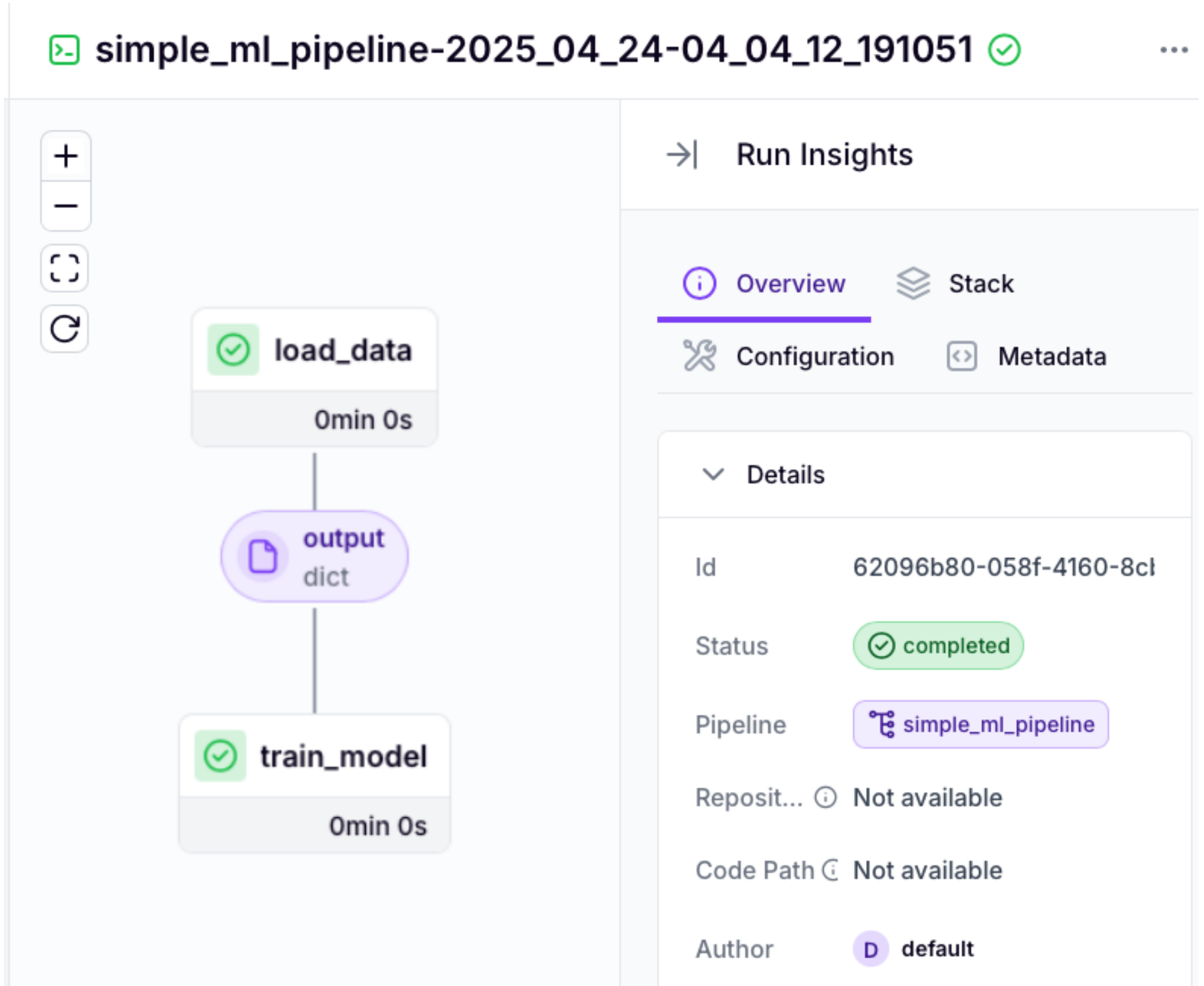
```
from zenml import pipeline, step

@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-4160-8cb
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

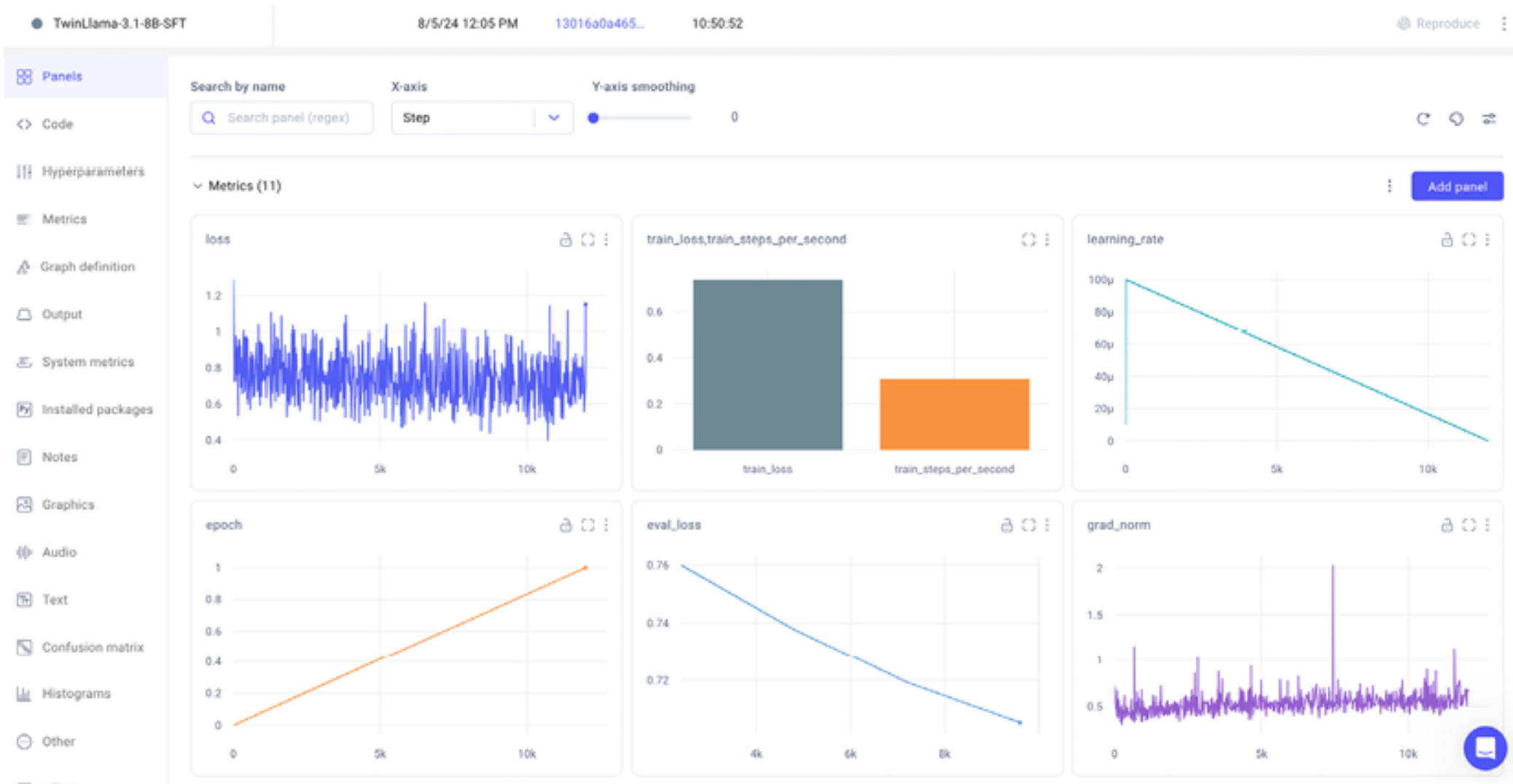
```
if __name__ == "__main__":
    hp_tuning_pipeline(step_count=4)()
```

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)
```