LLM Engineering & MLOps

A Comprehensive Study Guide on ZenML, AWS, and Infrastructure

Based on the LLM Twin Project

Prepared for Engineering Students

February 2, 2026

Contents

Chapter 1

ZenML: The MLOps Orchestrator

1.1 Introduction to ZenML

ZenML acts as the **bridge** between Machine Learning (ML) experimentation and MLOps production. In the lifecycle of an ML project, transitioning from exploratory notebooks to production-ready pipelines is a significant hurdle. ZenML addresses this by ensuring:

- **Traceability:** Knowing exactly what data and code produced a model.
- **Reproducibility:** The ability to rerun an experiment and get the same result.
- **Deployment:** Streamlining the path to production.

Core Concept: The Glue

ZenML is not just another platform; it is the "glue" that binds your infrastructure (AWS, GCP) with your code. It abstracts the infrastructure away, so your Python code doesn't need to know if it's running on a laptop or a SageMaker cluster.

1.2 The Concept of the Stack

ZenML introduces the concept of a **Stack**. A stack is a configuration that defines the infrastructure components used to run a pipeline. This allows you to switch environments (e.g., local vs. cloud) without changing your code.

A typical Stack consists of:

1. **Orchestrator:** Manages the workflow (e.g., Airflow, SageMaker, Vertex AI).
2. **Artifact Store:** Stores the inputs and outputs of steps (e.g., S3, GCS).
3. **Container Registry:** Stores Docker images (e.g., AWS ECR, Docker Hub).

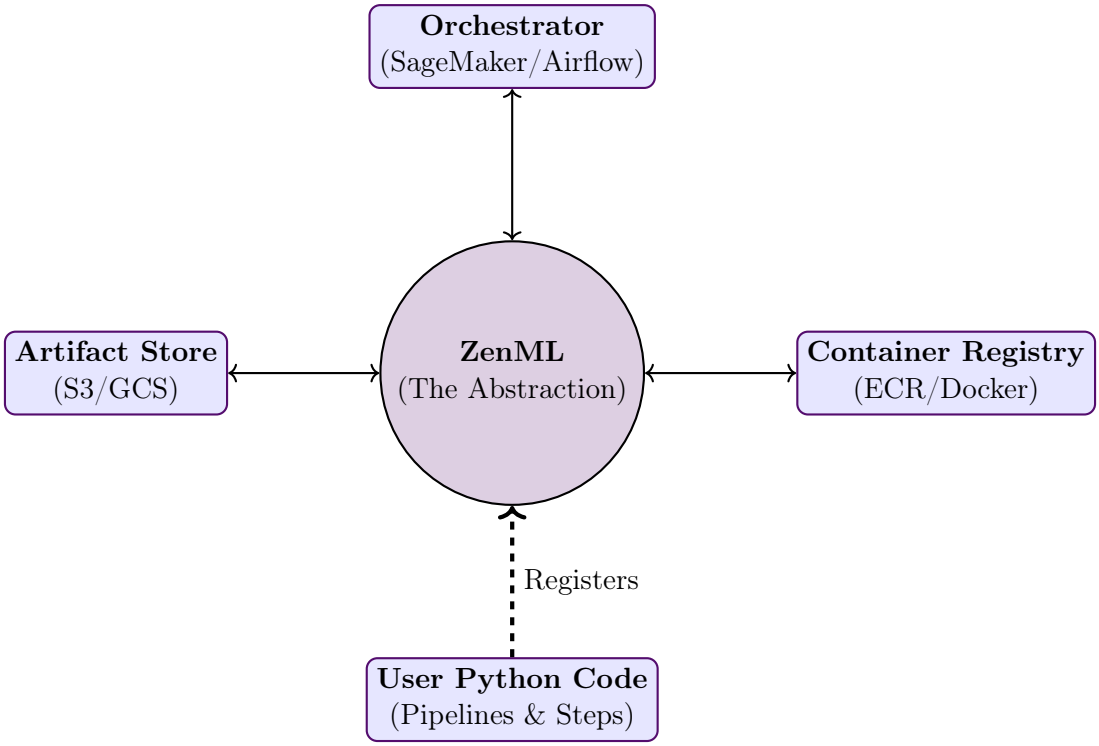

Figure 1.1: The ZenML Stack Architecture. Note how the User Code interacts with ZenML, which then interfaces with the infrastructure components.

1.3 Orchestrators, Pipelines, and Steps

An orchestrator automates and schedules your ML pipelines. In ZenML, this is achieved through two main decorators:

- `@step`: Decorates a function to make it a unit of execution.
- `@pipeline`: Decorates a function that defines the flow between steps.

1.3.1 Code Example: A Digital Data ETL

The following code demonstrates a pipeline used in the LLM Twin project to crawl data.

```
from zenml import pipeline
from steps.etl import crawl_links, get_or_create_user

@pipeline
def digital_data_etl(user_full_name: str, links: list[str]) -> None:
    # This acts as the manager calling the workers (steps)
    user = get_or_create_user(user_full_name)
    crawl_links(user=user, links=links)
```

Listing 1.1: Defining a Pipeline and Step

To run this pipeline using the configured toolchain (Poetry), one might use:

```
poetry poe run-digital-data-etl
```

1.3.2 Deep Dive: The Step

Steps must be modular. Each step should do one thing well. Below is the implementation of the ‘get_or_create_user‘ step.

```python
from loguru import logger
from typing_extensions import Annotated
from zenml import step
from llm_engineering.domain.documents import UserDocument

@step
def get_or_create_user(user_full_name: str) -> Annotated[UserDocument, "user"]:
    logger.info(f"Getting or creating user: {user_full_name}")
    # Logic to split name and find user in DB
    first_name, last_name = utils.split_user_full_name(user_full_name)
    user = UserDocument.get_or_create(first_name=first_name, last_name=
    last_name)
    return user
```

Listing 1.2: Step Implementation

Serialization Warning

ZenML must be able to serialize (pickle/save) the return value of a step to store it as an artifact. Standard types (int, str, list) work out of the box. Complex objects or custom types (like UUIDs) may require extending the ZenML Materializer.

Chapter 2

Artifacts, Metadata, and Configuration

2.1 Understanding Artifacts

In MLOps, an **Artifact** is any file produced during the ML lifecycle. This could be a dataset, a trained model, a plot, or logs.

Key properties of ZenML Artifacts:

1. **Versioning:** Every run produces a new version.
2. **Sharable:** Stored in the remote artifact store (e.g., S3).
3. **Metadata:** Describes the content without needing to download it.

2.2 Using Metadata

Metadata provides observability. Instead of downloading a massive dataset to check its size, you can view the metadata in the ZenML dashboard.

Below is an example of attaching metadata to a dataset generation step:

```python
from zenml import ArtifactConfig, get_step_context, step

@step
def generate_intruction_dataset(
    prompts: Annotated[dict, "prompts"]
) -> Annotated[
    InstructTrainTestSplit,
    ArtifactConfig(
        name="instruct_datasets",
        tags=["dataset", "instruct", "cleaned"],
    ),
]:
    datasets = ...  # (Logic to generate datasets)

    # Calculate metadata
    metadata = {
        "test_split_size": datasets.test_split_size,
        "categories": list(datasets.train.keys())
    }

    # Inject metadata into the context
    step_context = get_step_context()
    step_context.add_output_metadata(output_name="instruct_datasets", metadata=
    metadata)

    return datasets
```

Listing 2.1: Attaching Metadata to Artifacts

2.3 Runtime Configuration

Hardcoding parameters is bad practice in MLOps. ZenML allows injecting configuration via YAML files at runtime.

The YAML Config (e.g., for Maxime):

```yaml
parameters:
  user_full_name: Maxime Labonne
  links:
  - https://mlabonne.github.io/blog/posts/Finetune_Llama31.html
  - https://maximelabonne.substack.com/p/uncensor-any-llm
```

Loading at Runtime:

```python
config_path = root_dir / "configs" / etl_config_filename
run_args_etl = {
    "config_path": config_path,
    "run_name": f"etl_run_{dt.now()}"
}
# Using .with_options() to inject config
digital_data_etl.with_options()(**run_args_etl)
```

Chapter 3

Observability: Tracking Monitoring

3.1 Experiment Tracking: Comet ML

Training models is iterative. You need to compare "Run A" vs. "Run B".

Why Comet ML?

- **Hyperparameters:** Logs learning rate, batch size, etc.
- **System Metrics:** Tracks GPU/CPU/RAM usage (crucial for cloud costs).
- **Ease of Use:** Intuitive interface compared to MLflow or W&B.

3.2 Prompt Monitoring: Opik

Standard logging (printing text to console) fails for LLMs because LLM interactions are **chains** or **traces**.

The Challenge of LLM Logging

An LLM application often involves a chain: Prompt A $\rightarrow$ Output A $\rightarrow$ Prompt B (using Output A) $\rightarrow$ Final Result. You need a tool that can visualize this dependency tree (Trace) rather than just flat log lines.

Opik (by Comet) allows you to:

- Group multiple prompts into a single Trace.
- Debug specific steps in a chain.
- It is open-source and integrates well with the ZenML/Comet stack.

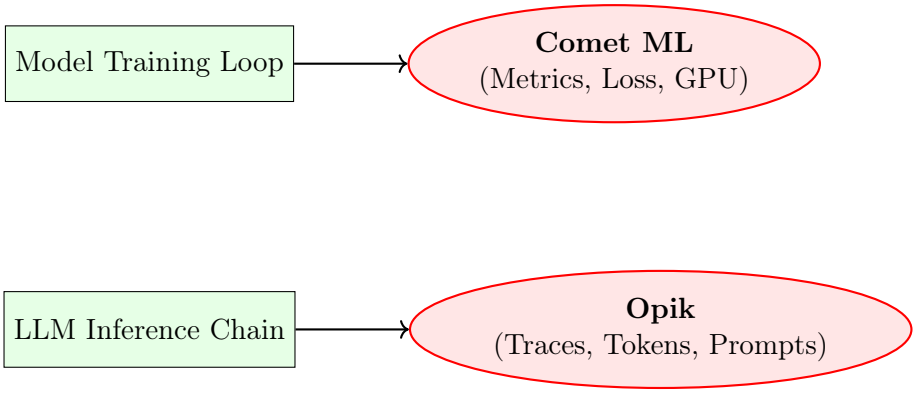

Figure 3.1: Separation of concerns: Comet ML for quantitative training metrics, Opik for qualitative LLM traces.

Chapter 4

Data Storage Infrastructure

4.1 The Database Layer

For the LLM Twin project, a dual-database approach is used to handle different stages of data processing.

4.1.1 MongoDB (NoSQL)

- **Purpose:** Storing raw, unstructured data collected from the internet (crawled HTML, text).
- **Why:** Schema flexibility. Web data is messy and changes structure often.

4.1.2 Qdrant (Vector Database)

- **Purpose:** Storing embeddings (numerical representations) of the processed text.
- **Why:** Optimized for similarity search (RAG - Retrieval Augmented Generation).
- **Efficiency:** Offers high performance (RPS, Latency) for GenAI applications.

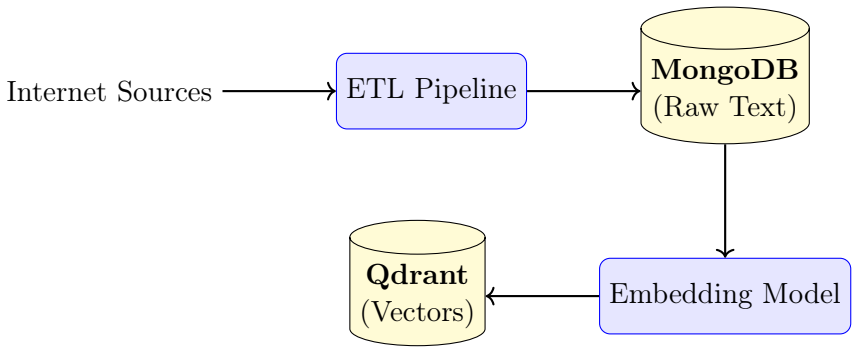

Figure 4.1: Data Flow: From unstructured web data to vector storage.

4.2 Cloud Infrastructure: AWS

The project uses AWS for its maturity and feature set.

4.2.1 AWS Components

- **S3 (Simple Storage Service):** Acts as the remote artifact store.
- **ECR (Elastic Container Registry):** Stores the Docker images required to run the pipeline steps.
- **SageMaker:** The compute engine. Used for:
  - Running Training Jobs (GPU clusters).
  - Hosting Inference Endpoints (REST API).

4.2.2 SageMaker vs. Bedrock

Decision Matrix

Bedrock: Serverless, pre-trained models. Good for quick prototyping, but less control.
SageMaker: Managed infrastructure. You control the model, the container, and the fine-tuning process. Chosen for this project to demonstrate engineering depth.

4.3 Setup Essentials

To interact with AWS programmatically (via ZenML or CLI), you need:

1. **IAM User:** Created with 'AdministratorAccess' (for learning purposes) or least-privilege (for production).
2. **Access Keys:** `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY`.
3. **Configuration:** Run `aws configure` to set credentials and region (default: `eu-central-1`).

Chapter 5

Summary & Next Steps

5.1 Architecture Recap

We have built an architecture that is:

1. **Vendor Agnostic:** Thanks to ZenML Stacks.
2. **Reproducible:** Thanks to Artifacts and Versioning.
3. **Observable:** Thanks to Comet ML and Opik.
4. **Scalable:** Thanks to AWS SageMaker and Qdrant.

5.2 Future Roadmap

With the infrastructure in place, the next steps in an LLM Engineering lifecycle would be:

- **Data Engineering:** Refining the cleaning processes in the ETL pipeline.
- **RAG Implementation:** Connecting Qdrant to the LLM for context retrieval.
- **Fine-tuning:** Using SageMaker to train the LLM on the specific datasets generated.

End of Study Guide