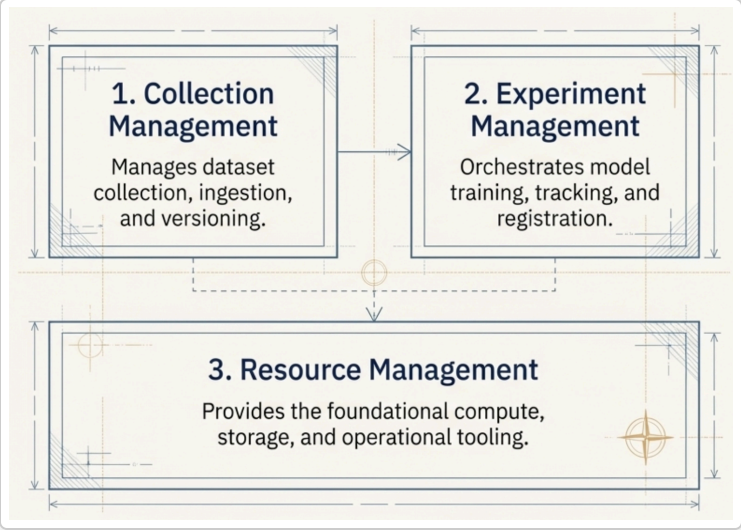
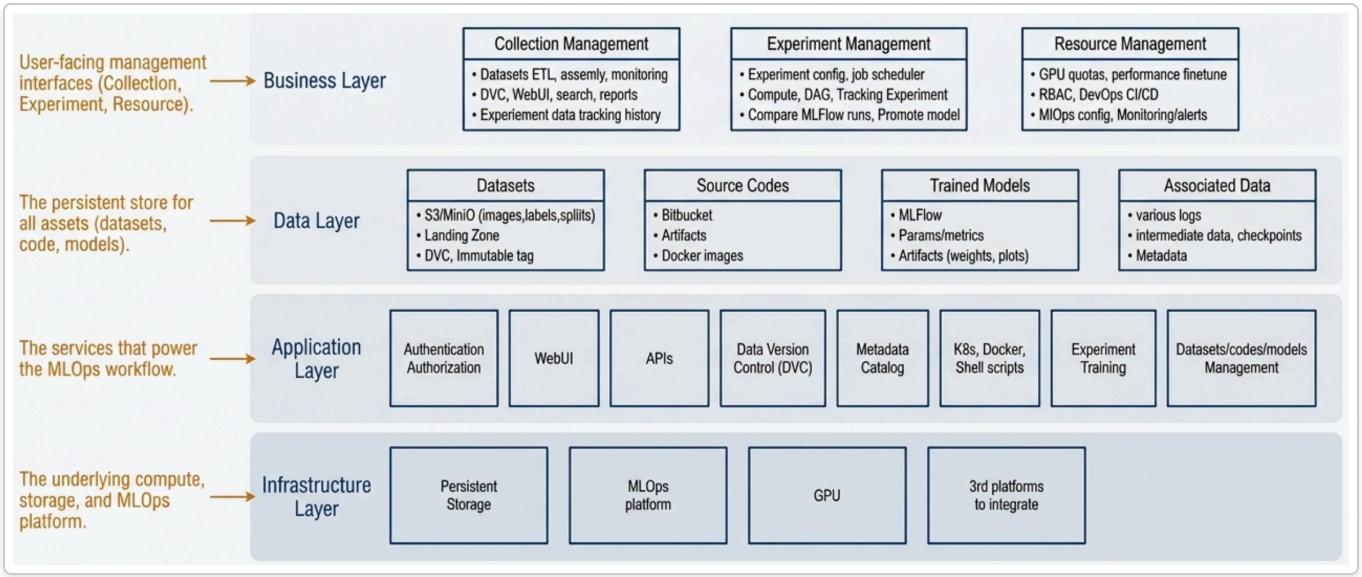
Autonomous Driving Training Platform – MLOps Architecture & Workflow Details

This document provides an in-depth look at the **architecture and workflows** of the autonomous driving model training platform. It breaks the system into three major subsystems:

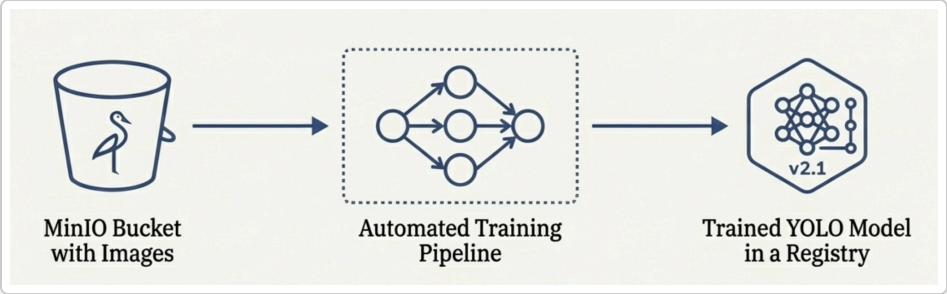
1. **Collection Management**,
2. **Experiment Management**
3. **Resource Management**



Each subsystem integrates with its underlying components to deliver specialized features, while coordinating seamlessly with others to support the overall workflow:

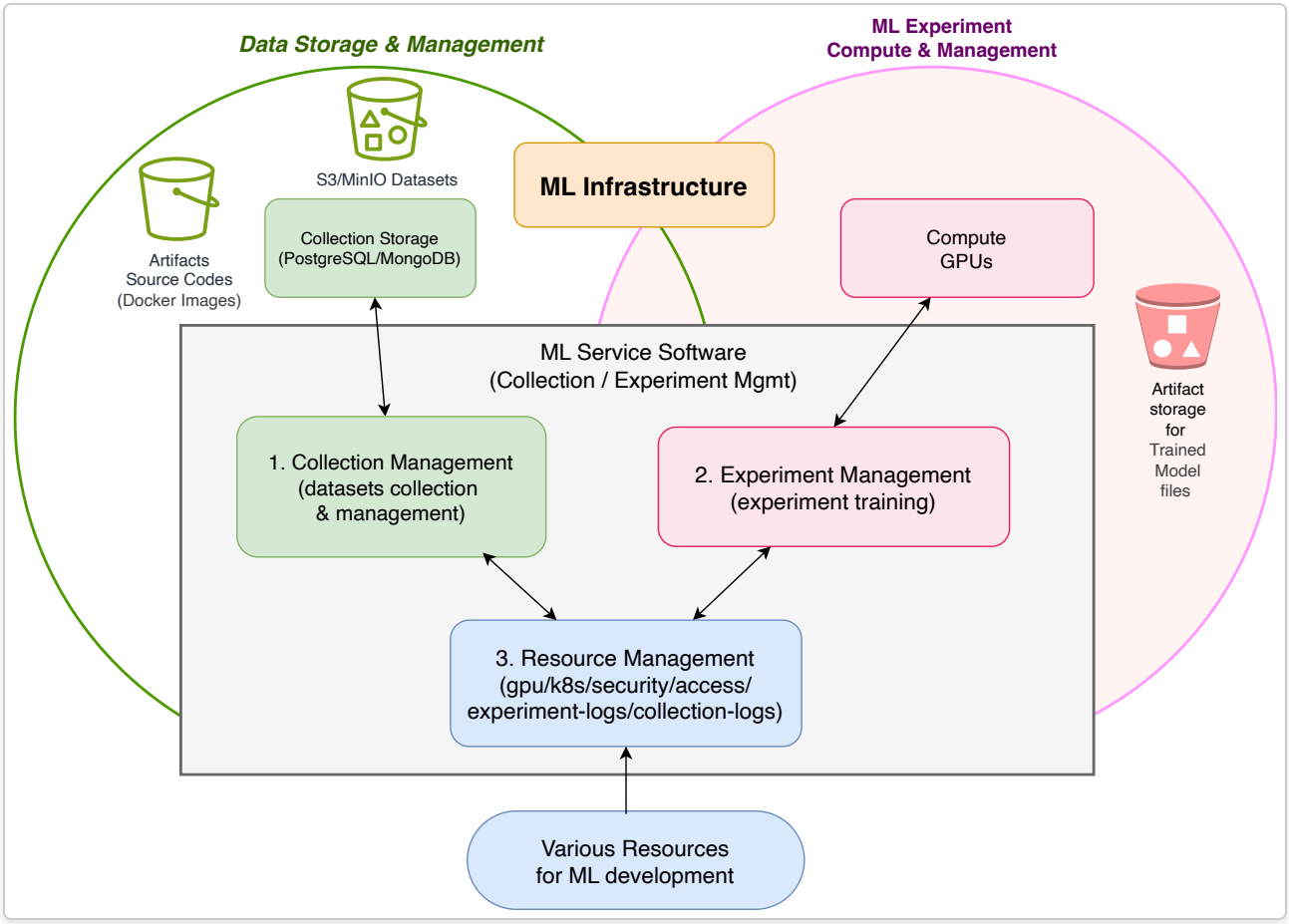


Each section details components, data flows, and operational responsibilities. The architecture spans **data ingestion pipelines**, **training experiment orchestration (development and production)**, and the **infrastructure & DevOps tooling** that supports the platform.



High-Level Architecture Overview

Figure: High-level architecture layers of the ML training platform



The system is organized into layers for **data storage**, **experiment compute**, and **service software**, aligned with three focus areas: Collection Management, Experiment Management, Resource Management

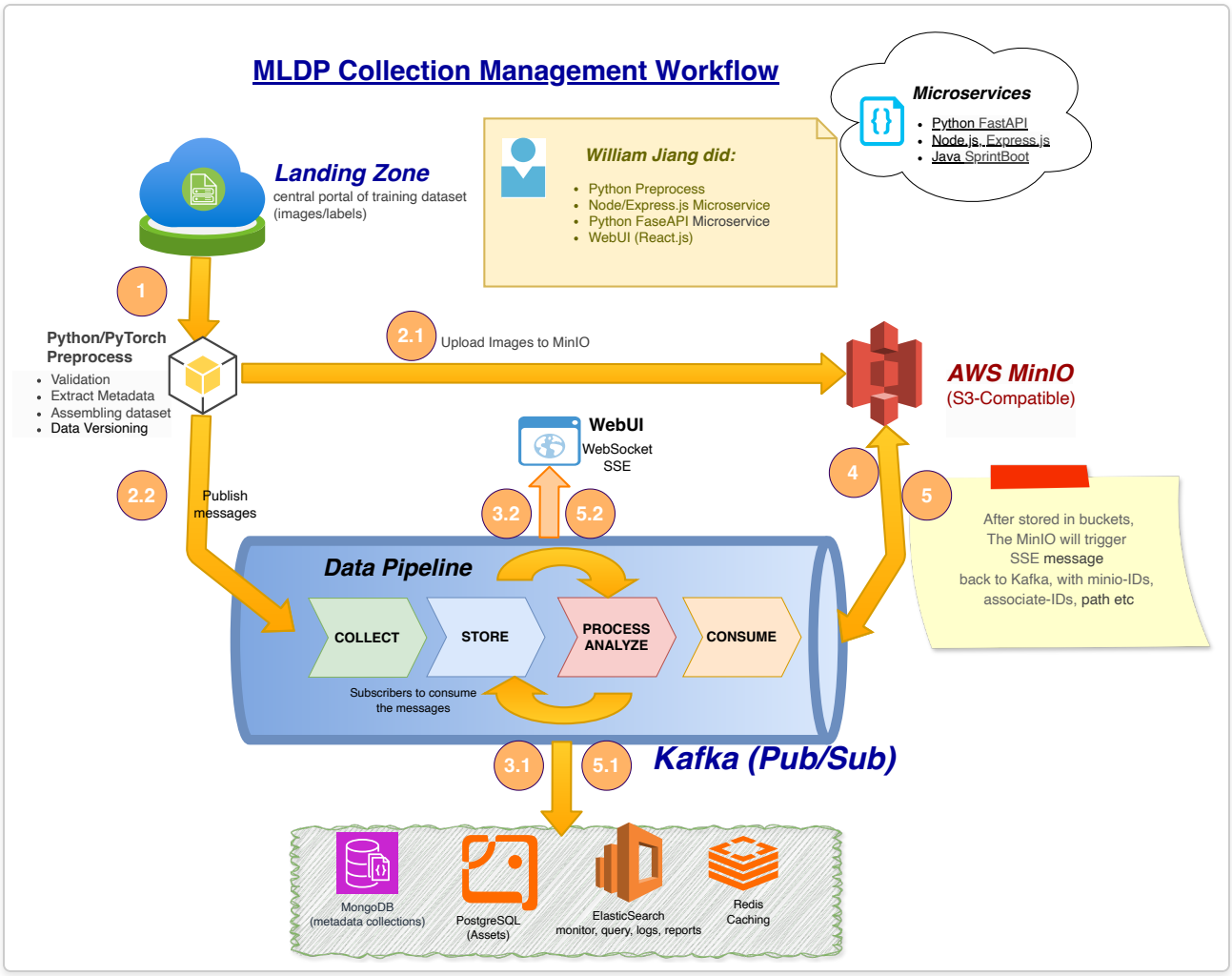
Core data stores (S3/MinIO, databases), compute resources (GPU cluster), and software services (collection & experiment management APIs) are depicted in context.

---

1. Collection Management

This subsystem is responsible for **data collection, ETL processing, and dataset management**. The goal is to transform raw image data and annotations into **curated, versioned datasets** ready for model training.

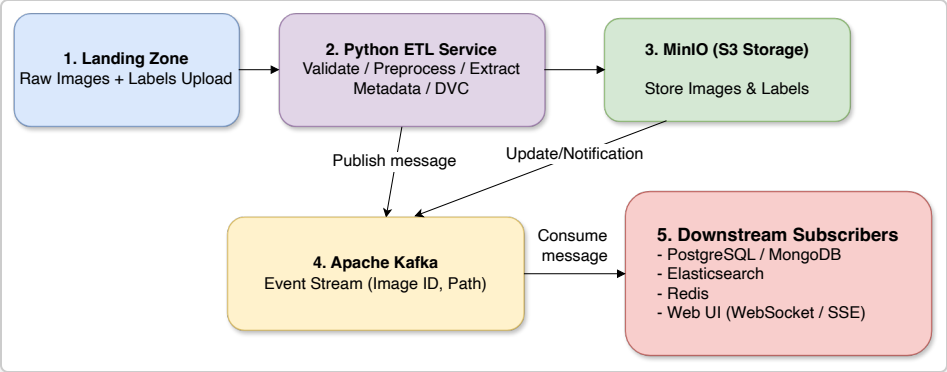
Data Ingestion Workflow



- The entry point is the **Landing Zone**, a centralized portal (analogous to an internal “Google Drive”) where data engineers upload new driving images and labels.
- The Landing Zone acts as a **staging area**, structured by user, folder, and timestamp.
- A **Python-based ETL service** continuously or periodically monitors this area for new files.
- When new images are detected, the service:
  - Transfers files to **AWS S3–compatible object storage (MinIO)**

    - Performs image format validation
    - Executes preprocessing (e.g., PyTorch-based normalization or augmentation)
    - Extracts metadata (timestamps, sensor info, label counts)
- Once stored in MinIO:
    - **Server-Side Event (SSE)** notifications are generated per object
    - Events are published to an **Apache Kafka** topic with image IDs and storage paths

Figure: A close look at dataset movement



1. Upload to Landing Zone: Data Engineers prepare and upload
2. ETL processing and validation: Python multiprocess, Pytorch
3. Storage in MinIO with SSE notifications: Datasets processing
4. Publish metadata services via Kafka Pipeline
5. Microservices consume these metadata for association, versioning, and later history tracking and reports.

Python Automated ETL & Preprocessing

The Python-based ETL service orchestrates the data ingestion pipeline through five key stages:

- **Collection** – Monitors Landing Zone for new uploads via file system polling or event triggers
- **Validation** – Verifies image formats, checks annotation schemas, and detects corrupted files
- **Storage** – Transfers validated data to MinIO with integrity checks (MD5/SHA256 hashing)
- **Indexing** – Extracts metadata (timestamps, sensor info, label counts) for downstream services
- **Versioning** – Records dataset versions via **DVC (Data Version Control)** for reproducible experiments

The service is implemented using **Python** with **FastAPI** for RESTful endpoints and **PyTorch** for image preprocessing tasks such as normalization, resizing, and augmentation.

Kafka Consumers & Metadata Handling

Multiple microservices subscribe to Kafka events:

- **Metadata Service** → stores image metadata and labels in **MongoDB**
- **Asset Management Service** → updates dataset and asset inventory in **PostgreSQL**
- **Search Indexer** → updates **Elasticsearch** for search and analytics
- **Caching Layer** → **Redis** accelerates access to frequently used metadata

Kafka's pub/sub model enables **scalability and extensibility**, allowing new consumers (e.g., automated training triggers) without disrupting upstream flows.

Dataset Collections & Versioning

- Data is organized into **collections**, representing dataset groupings (e.g., *"Highway driving images – Summer 2025"*).
- A collection includes:
  - References to images and labels in MinIO
  - Metadata (owner, category, creation date)
- **DVC (Data Version Control)** versions collections and tracks lineage.
- Each update records:
  - Dataset version
  - MinIO file hashes
  - Associated metadata

By the end of ingestion, data is **validated, stored, cataloged, indexed, and versioned** for reproducible experiments.

Collection Management UI & Features

- Web-based UI built with **React (frontend)** and **Node.js/Express (backend)**
- Integrates with backend services (e.g., Python FastAPI)
- Enables users to manage:
  - **Assets** – images and labels with previews and metadata
  - **Collections** – dataset groupings and ownership
  - **Jobs** – ingestion and transfer records

Real-Time Monitoring & Reporting

- Live ingestion progress via **WebSockets / SSE**
- Real-time status updates, logs, and progress bars
- Reporting dashboards:

- Images ingested over time
- Success/failure rates
- Pipeline health metrics (via Elasticsearch)

Collection Management delivers a robust pipeline to **collect, store, index, version, and monitor training data**, with a user-friendly interface for full visibility.

---

2. Experiment Management

Experiment Management handles **model training workflows**, from orchestration to tracking and model governance. Two modes are supported:

- **Development**: Apache Airflow + Kubernetes
- **Production**: Kubeflow Pipelines (KFP)

Figure: Development - fast-pace experiment iteration

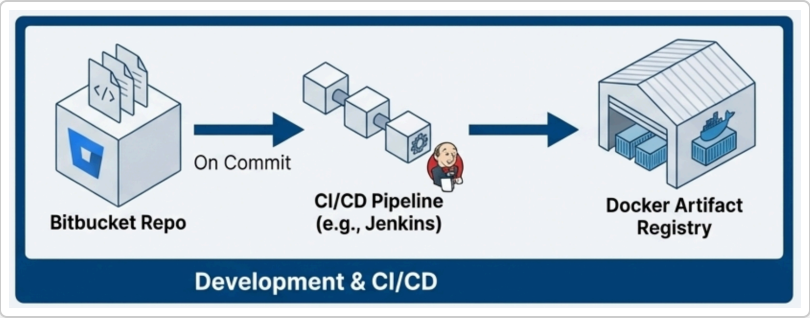
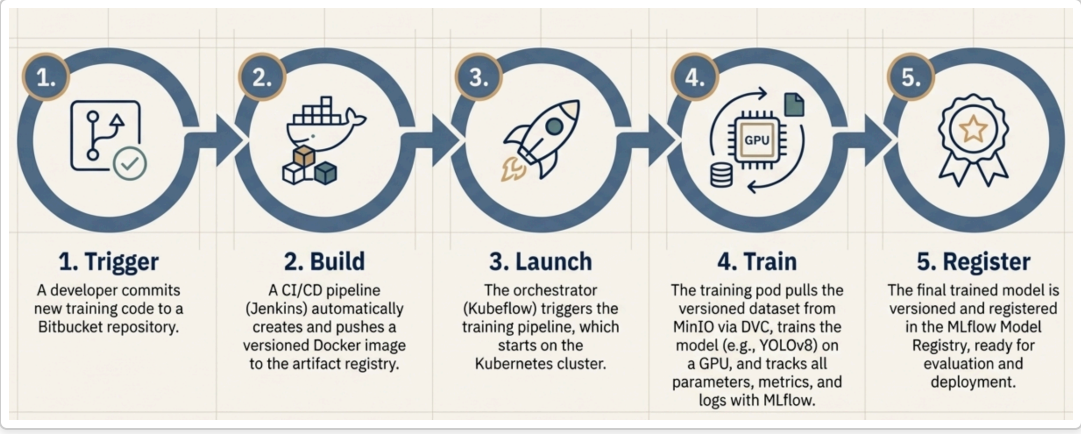


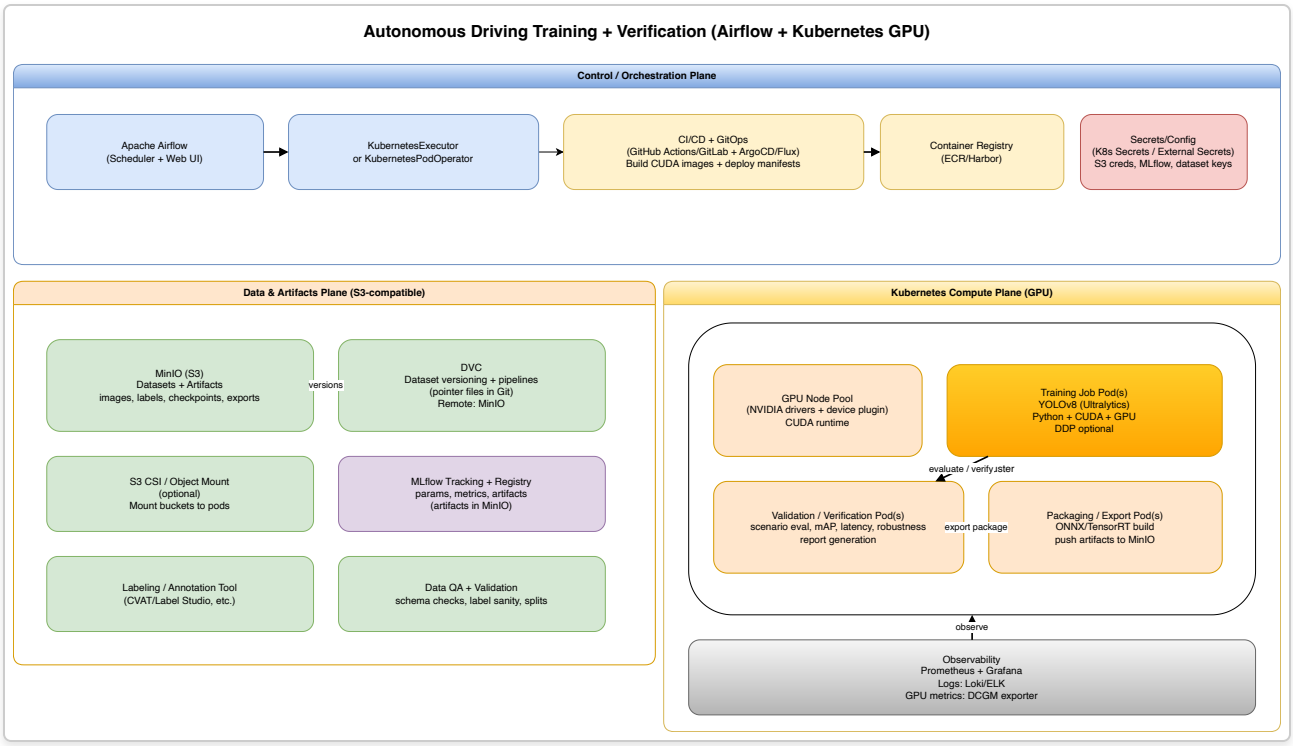
Figure: Production - End-to-end experiment workflow

The following is an automated training run using Kubeflow in production.



Development Pipeline (Airflow + Kubernetes)

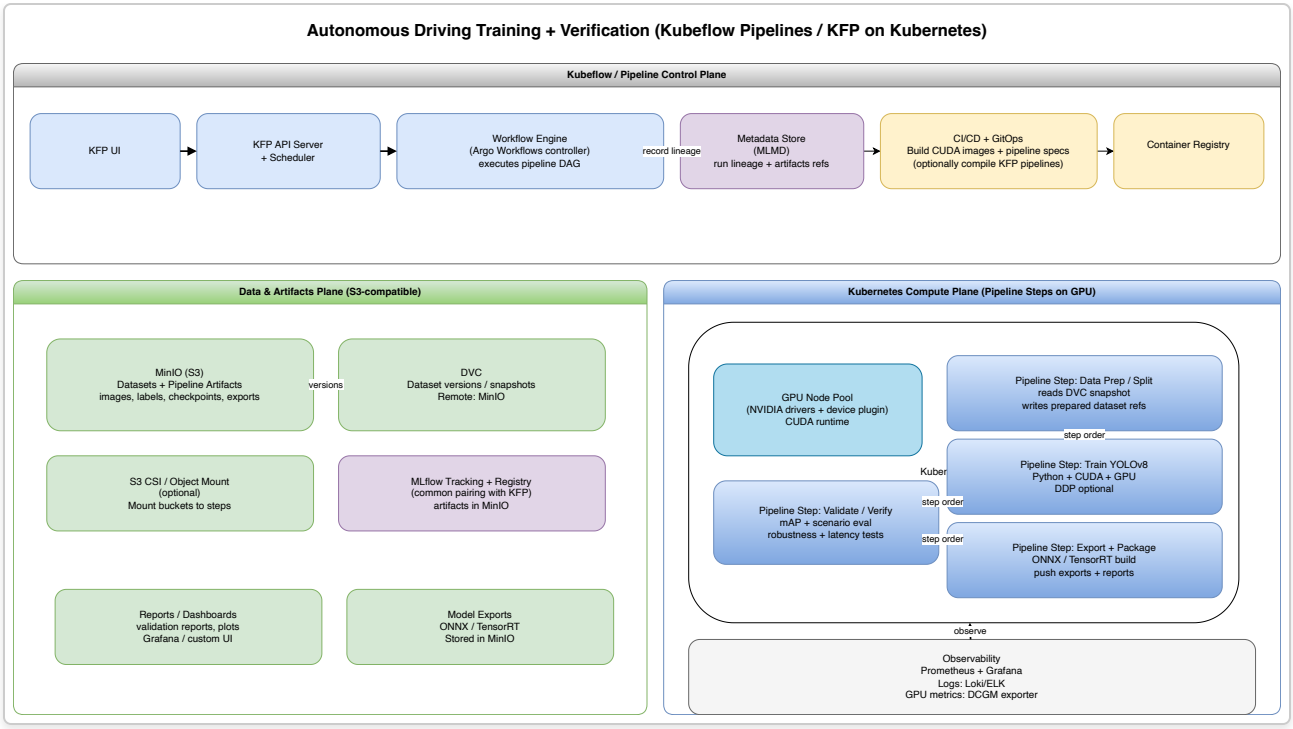
Figure: Airflow Workflow



- **Airflow DAGs** schedule and coordinate the ML workflow, including: Data preparation, Model Training, Evaluation, Model export
- **Kubernetes Pod Operator** launches containerized training jobs on GPU-enabled nodes.
- Models **YOLOv8** (e.g., for detecting cars, pedestrians, etc.) are trained using **Python / PyTorch** on Kubernetes GPUs.
- Training metrics and parameters are logged to an **MLflow tracking server**, providing:
  - Experiment tracking
  - Model artifact management via the **MLflow Model Registry**
- Training containers:
  - Use pre-built Docker images
  - Pull datasets from MinIO (optionally via DVC)
  - Train models (e.g., YOLOv8)

Experiment Tracking

Figure: Kubeflow Workflow



- **MLflow** logs: Metrics and hyperparameters, Artifacts (e.g., YOLOv8 `.pt` files)
- Trained models are registered (e.g., *YOLOv8 – v2*).
- Logs and outputs are forwarded to **Elasticsearch** and visualized via **Kibana**.

Trade-off: Airflow provides flexibility but is not ML-native, motivating a production shift to Kubeflow.

Development vs. Production Orchestration

- **Airflow**: General-purpose orchestration, ideal for rapid experimentation and ad-hoc workflows
- **Kubeflow Pipelines**: ML-native orchestration, optimized for lifecycle management, reproducibility, and production-scale workflows

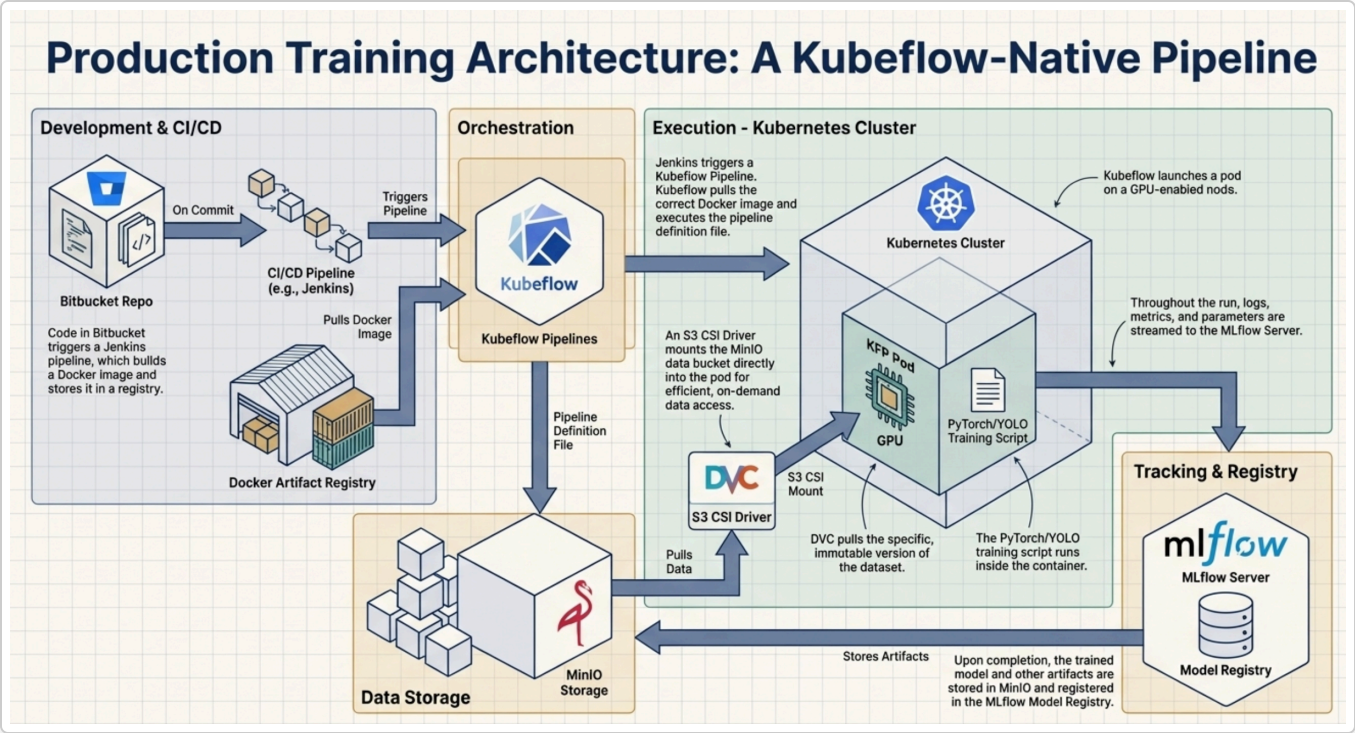
Aspect	Development (Apache Airflow)	Production (Kubeflow Pipelines)
Orchestration	General-purpose workflow engine (data engineering, ETL, experimentation)	ML-native lifecycle management on Kubernetes
Workflow Definition	Python DAGs trigger Kubernetes Pod Operators	Containerized pipeline components with dependency graphs
Data Access	Manual DVC pulls or custom scripts	S3 CSI Driver for efficient bucket mounting

Aspect	Development (Apache Airflow)	Production (Kubeflow Pipelines)
Execution Layer	KubernetesPodOperator launches isolated pods	Native Kubeflow pipeline pods with optimized scheduling
Experiment Tracking	MLflow (manual logging)	MLflow (automatic logging)
CI/CD Integration	Jenkins builds images; manual DAG triggers	Jenkins triggers pipelines via KFP API
Artifact Storage	MinIO (S3-compatible)	MinIO (S3-compatible)
Monitoring	Prometheus / Grafana	Prometheus / Grafana + Kubeflow UI
Model Registration	Manual review and registration	Automatic MLflow Model Registry integration

Airflow enables fast iteration; Kubeflow provides **production-grade, ML-optimized pipelines**.

Production Pipeline

Figure: Kubeflow Pipelines workflow



- **Kubeflow Pipelines (KFP)** orchestrate end-to-end ML workflows on Kubernetes.
- Pipelines consist of containerized components:
  - Data preparation

    - Training
    - Evaluation
    - Model registration/export

CI/CD Integration

- Code merge in Bitbucket triggers **Jenkins**:
    - Builds versioned Docker images
    - Pushes images to the registry
    - Invokes KFP via API
- Each run is tied to:
    - Specific code commit
    - Dataset version
    - Container image

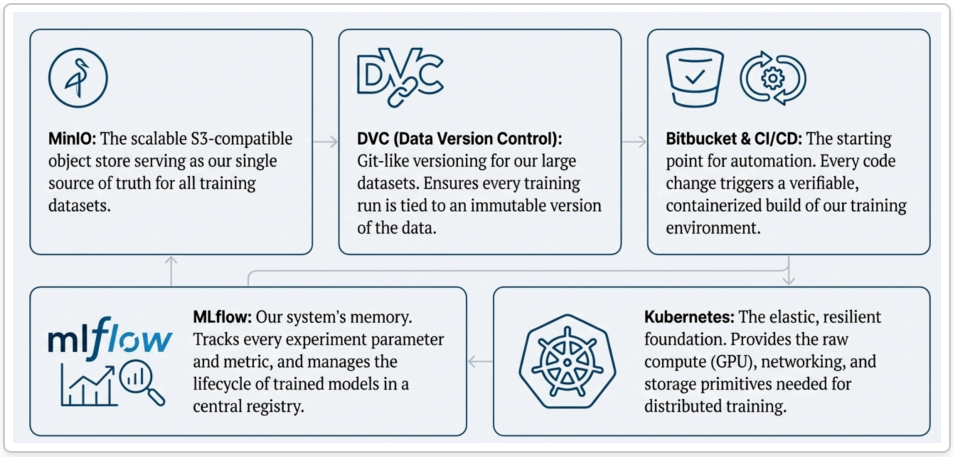
Data Access & Training

- **S3 CSI Driver (Container Storage Interface)** mounts MinIO buckets into pods as local volumes.
- Enables on-demand data streaming without copying the full dataset—critical for minimizing pod startup time and reducing storage overhead.
- Training runs on **NVIDIA GPUs** with dedicated GPU nodes.
- Metrics are visible via:
    - MLflow
    - Kubeflow UI
    - Prometheus/Grafana

Model Registration

- Final steps:
    - Store model artifacts
    - Log results to MLflow
    - Register model in **MLflow Model Registry**
- Optional downstream handoff for deployment.

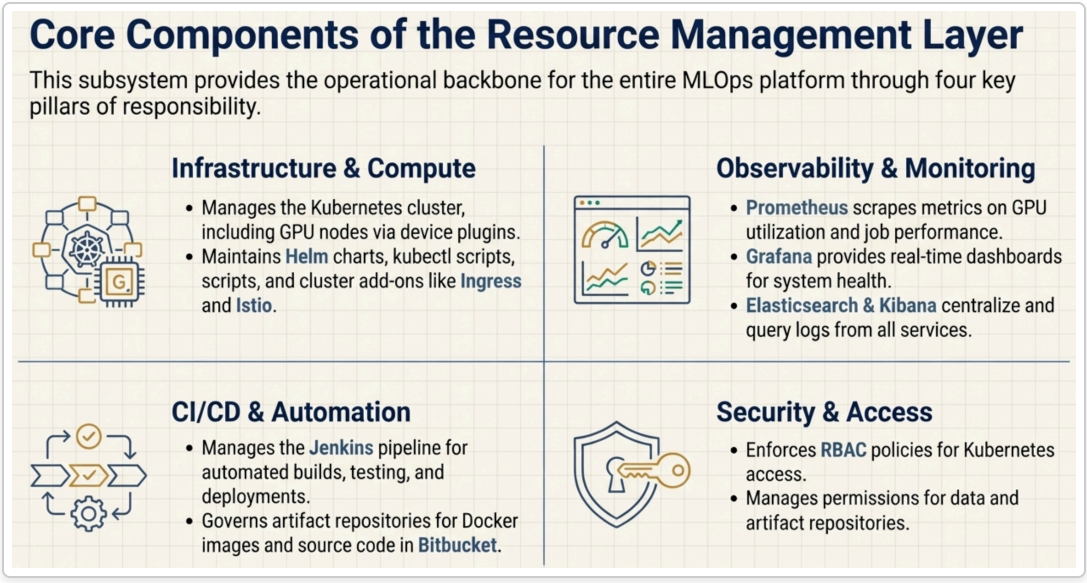
Core components



---

3. Resource Management

Resource Management governs **infrastructure, compute, storage, observability, security, and DevOps**, forming the backbone of the platform.



Component	Description
Kubernetes Cluster	GPU nodes for training workloads
Control Plane	Kubernetes control plane, Helm charts, `kubectl` scripts
Platform Services	Kafka, Kubeflow, MLflow, APIs
Ingress Controllers	Nginx, Istio - route traffic to UIs and APIs
RBAC & Access Control	Enforce permissions for namespaces, secrets, data access, experiment execution

GPU Management

Aspect	Details
Nodes & Pods	Nodes & Pods versions, operating engineers, categories views
Scheduling	Policies for efficient GPU utilization
Monitoring	Capacity planning and prioritization

Data Storage Management

Storage Type	Technology	Management Focus
Object Storage	MinIO	Capacity, throughput, backups, replication
Document Database	MongoDB	Backup, Query
Relational Database	PostgreSQL	Backup, Query, performance tuning

Observability & Monitoring

Tool	Purpose	Monitored Components
Prometheus	Metrics collection	Kubernetes nodes/pods, GPUs, Kafka, MinIO, databases
Grafana	Visualization dashboards	Training throughput, resource utilization, pipeline health
Elastic Stack	Centralized logging	Collection pipelines, training jobs, platform services
Alerting	Failure notification	Teams notified of failures or bottlenecks

CI/CD, Artifact Management & DevOps

Component	Technology	Responsibilities
Source Control	Bitbucket	Code repository management
CI/CD Pipeline	Jenkins	Run tests, build Docker images, deploy via Helm/kubectl
Artifact Registry	Container Registry	Store service images, training images, supporting binaries
Web Server	Nginx	Configuration and management
ML Platform	MLflow	Server maintenance
Governance	Policies & Scripts	Resource cleanup, retention policies

Resource Management ensures **stability, observability, security, and automation** across the platform. By managing compute, storage, CI/CD, and monitoring holistically, it enables data scientists and engineers to focus on model development with confidence in the underlying infrastructure.

Summary of Subsystem Integration

The three subsystems collaborate through well-defined integration points:

Integration	Mechanism	Purpose
Collection → Experiment	DVC-versioned datasets pulled from MinIO	Provides training data for experiments
Collection → Resource	Kafka events monitored by Prometheus	Tracks ingestion metrics and pipeline health
Experiment → Resource	MLflow server hosted on Kubernetes	Centralizes experiment tracking and model registry
Experiment → Collection	Training metadata API callbacks	Links trained models back to source datasets

Figure Kuberflow orchestration with Kubernetes

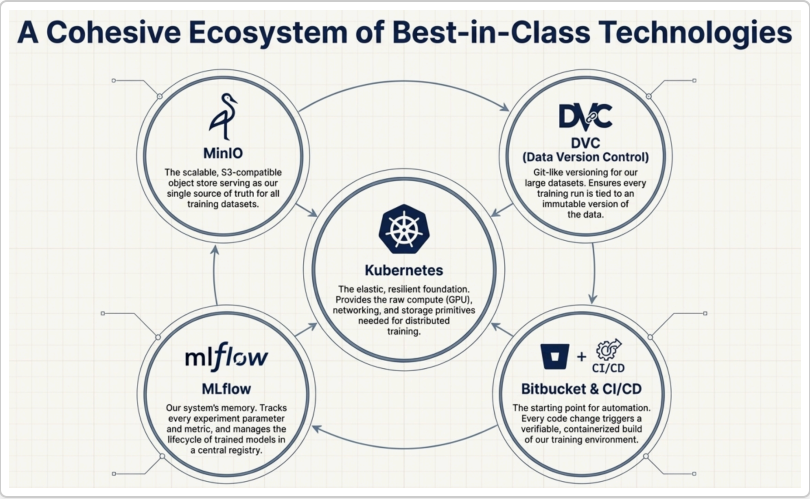
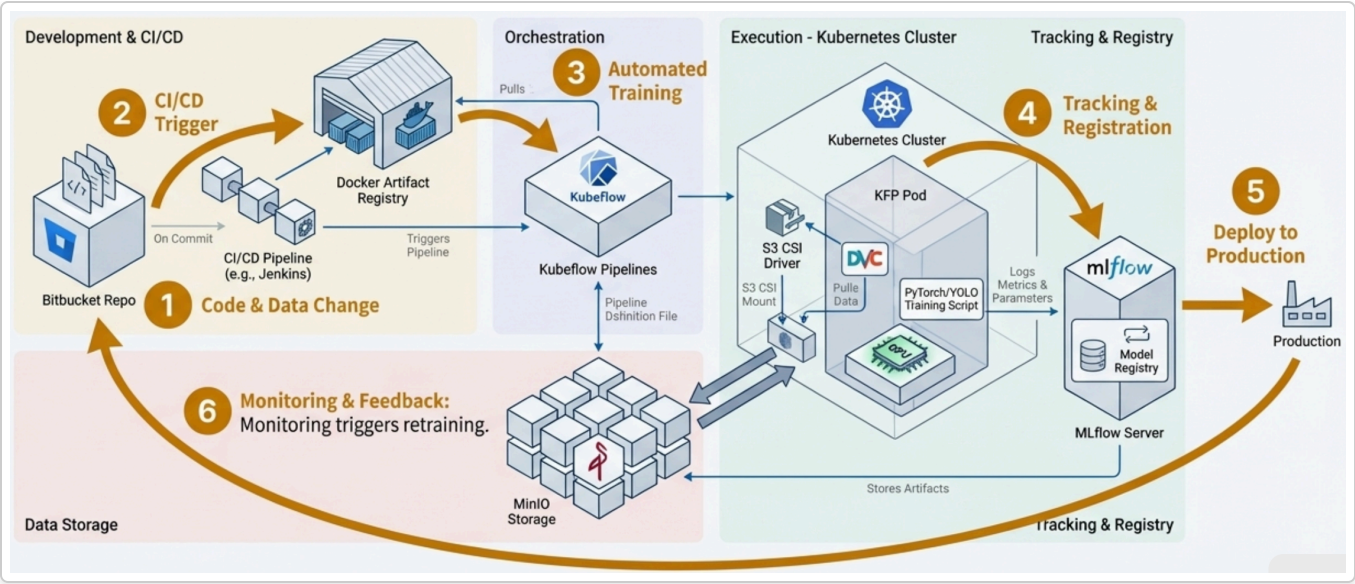


Figure The Experiment Tracking Training Loop



What William Jiang contributed

I worked across **all three subsystems** of the platform, with primary focus on **Collection Management**, followed by **Resource Management**, and supporting involvement in **Experiment Management**.

---

1. Collection Management (Primary)

- Built **Python ETL pipelines** for large-scale image and label ingestion
  - PyTorch-based preprocessing
  - Multiprocessing for performance
  - Dataset versioning with **DVC**
- Developed **backend APIs** (FastAPI, Node.js) to support:
  - Dataset ingestion (Landing Zone)
  - Metadata and report generation
  - Linking experiment results back to datasets
- Implemented the **Collection Management Web UI** (React.js)
  - Dataset browsing and management
  - Real-time status updates using **WebSocket / SSE**
  - Kafka event consumption

---

2. Resource Management

- Implemented **backend APIs** for resource visibility and reporting
- Managed **Kubernetes and Docker-based infrastructure**
  - CI/CD pipelines with **Jenkins**
  - Deployments via `kubectl` and Helm
- Set up **monitoring and observability**
  - Metrics with **Prometheus**
  - Dashboards with **Grafana**

---

3. Experiment Management (Supporting)

- Integrated **Kubeflow Pipelines** for production training workflows
- Used **MLflow** for experiment tracking and model registry
- Supported experiment log querying and result analysis

Glossary

Term	Definition
CSI	Container Storage Interface - Kubernetes storage plugin standard
DAG	Directed Acyclic Graph - Airflow workflow definition
DVC	Data Version Control - Git-like versioning for datasets
KFP	Kubeflow Pipelines - ML workflow orchestration framework
SSE	Server-Side Events - HTTP streaming for real-time updates