A Technical Blueprint for a Humanitarian Geospatial MLOps Platform

Part I: Architectural Blueprint for a Humanitarian Geospatial MLOps Platform

1. Conceptual Framework and Principles

This section establishes the foundational principles of the platform, aligning modern Machine Learning Operations (MLOps) practices with the unique demands of the humanitarian sector. The architecture is designed not only for technical efficiency but also to foster trust, transparency, and accessibility which is crucial in high-stakes environments such as disaster response and sustainable development monitoring.

1.1 Mission

The application of machine learning to geospatial data holds immense promise for humanitarian organizations. From identifying damaged infrastructure after an earthquake to monitoring deforestation or tracking the expansion of refugee camps, AI can provide timely insights at an unprecedented scale. However, moving from a research-grade model to a reliable, production-level system presents significant operational challenges. MLOps, a discipline that combines ML development with IT operations, provides a structured approach to overcoming these hurdles by automating and standardizing the model lifecycle.
For this platform, MLOps is not an end in itself but a means to achieve specific humanitarian objectives. The system must be guided by a set of core principles that differentiate it from purely commercial MLOps platforms:

- **Transparency and Auditability:** Every prediction must be traceable back to the exact model version, training data, and source code that produced it. This is critical for accountability and for understanding a model's limitations when making life-affecting decisions.
- **Reproducibility:** Any experiment or training run must be perfectly reproducible by another user or on another system. This principle is fundamental to scientific validation and collaborative development.
- **Accessibility:** The platform should be usable by individuals with varying levels of technical expertise. While data scientists will develop the models, field experts

and analysts must be able to easily discover, run, and interpret the outputs of these models without needing to be MLOps specialists.
- **Cost-Effectiveness:** Humanitarian organizations often operate under significant budget constraints. The architecture must prioritize open-source technologies and efficient resource utilization to minimize operational costs.

To realize these principles, the architecture is built around the concept of the "Model as a Standardized Unit" Each machine learning model contributed to the platform is treated as a self-contained, discoverable, and executable package. This package encapsulates not only the model weights but also its code, dependencies, execution environment, and descriptive metadata. This approach ensures that any model registered in the system can be reliably trained, tested, and deployed through standardized, automated workflows.

1.2 MLOps Maturity Model

MLOps maturity models provide a framework for assessing and improving an organization's capabilities in deploying and managing machine learning systems. Several models exist, including those from Google, Microsoft Azure, and Gartner, which generally describe a progression from manual processes to fully automated, continuously improving systems.
For this humanitarian platform, a pragmatic adaptation of these models is necessary. The goal is not to achieve the highest possible level of automation for its own sake, but to implement a level of automation that maximizes reliability and trust while retaining critical human oversight.

- **Level 0: Manual Process.** This is the baseline state, characterized by disconnected, script-driven, and interactive processes. Data scientists manually train models, and hand over artifacts (like a model weight file) to engineers for deployment. This process is error-prone, lacks reproducibility, and is difficult to scale. The proposed platform is designed to move decisively beyond this level.
- **Level 1: ML Pipeline Automation.** This level introduces the automation of the model training pipeline. Instead of manual steps, a workflow orchestrator automatically retrains the model when triggered, producing a validated and versioned model artifact. This is a core feature of the proposed architecture, enabling continuous learning to test and be able to run models with different training data.
- **Level 2: CI/CD Pipeline Automation.** At this level, a full Continuous Integration/Continuous Deployment (CI/CD) system is in place. Changes to the model's source code automatically trigger a pipeline that tests the code, retrains

the model, validates its performance, and deploys it to production. This represents a highly mature and efficient state.

- **Humanitarian Adaptation (Level 3+):** While the platform will automate the *mechanics* of Level 2, it will deliberately introduce a manual gate for model promotion. In a humanitarian context, the consequences of a model behaving unexpectedly on new data can be severe. A model retrained on satellite imagery from a new region or a different season might learn unforeseen biases or fail in subtle ways.

Therefore, the platform's workflow will ensure that while a new model version can be automatically trained and evaluated, its promotion to the "production" stage for use by end-users requires explicit approval from a human domain expert (admin ?). This "human-in-the-loop" governance is a deliberate design choice, not a technical limitation.

2. Architecture Overview

The platform's architecture is designed as a cohesive ecosystem of open-source components, orchestrated to support three primary workflows: ***Model Registration, Automated Training, and On-Demand Inference.*** The following diagram provides a visual representation of the system, illustrating the flow of data and control signals between its core components.

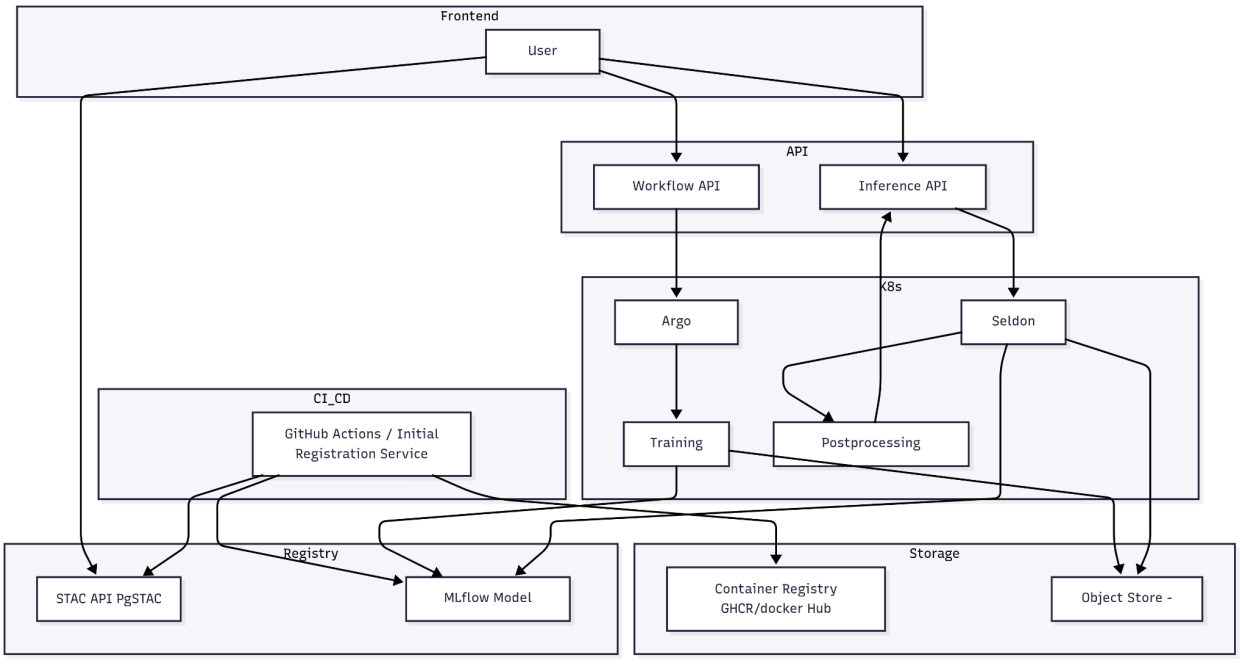
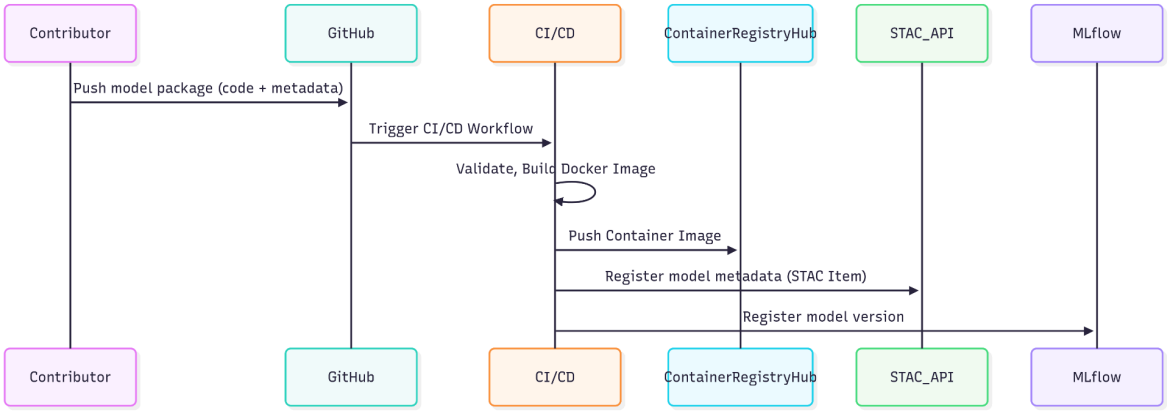
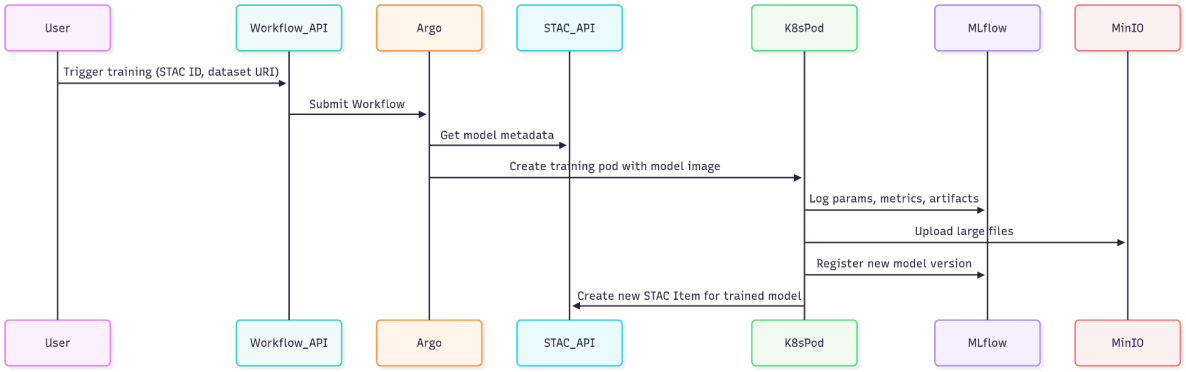


Figure 1: A high-level architectural diagram showing the core components and their interactions across the three main workflows. The flow of control and data is illustrated for model registration, training, and inference.

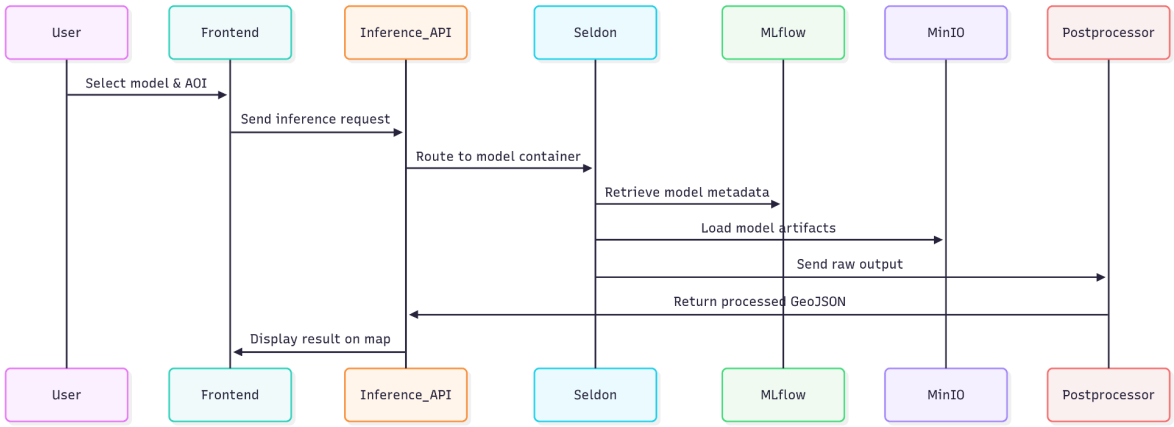
The architecture is centered around a Kubernetes cluster, which provides the scalable and reproducible execution environment for all containerized tasks. The key interactions and workflows are described below.



- **Model Registration Workflow:** This workflow onboards a new model into the platform.
    1. A Data Scientist or contributor prepares a model package, which includes the model's source code, MLProject file defining entry points for training preprocessing postprocessing and inference, a Dockerfile defining its environment, and a STAC MLM (Machine Learning Model) metadata file in JSON format.
    2. The package is submitted to the platform via a Git repository or API endpoint.
    3. A CI/CD Pipeline (e.g., GitHub Actions) or an initial registration service is triggered. It validates the package, builds the Docker container image, and pushes it to a Container Registry (e.g., Docker Hub, Github CR).
    4. The pipeline then makes an API call to the STAC API (with PgSTAC) to register the model's metadata, creating a new STAC Item that describes the model and includes a link to its container image.
    5. Simultaneously, the model is registered in the MLflow Model Registry, creating an initial version.



- **Training Workflow:** This workflow is triggered to train a new version of an existing model.
    1. An Administrator or an automated process initiates a training job via an API call to the Workflow API Gateway. The request specifies the STAC ID of the model to be trained and the location of the new training data (e.g., a URI to a DVC-tracked dataset).
    2. The API Gateway triggers an Argo Workflow.
    3. The first step in the workflow queries the STAC API to retrieve the model's container image URI and other essential metadata.
    4. Argo then orchestrates the creation of a Kubernetes Pod. This pod uses the specified container image, ensuring the correct environment is used for training.
    5. The command executed within the pod is mlflow run., which invokes the MLflow Project packaged within the container. The workflow passes the training data location as a parameter.
    6. During execution, the training script communicates with the MLflow Tracking Server, logging all parameters, metrics, and artifacts. Model weights and other large artifacts are stored in the MinIO Object Storage bucket designated for MLflow.
    7. Upon successful completion of the training run, a final workflow step registers the newly trained model as a new version in the MLflow Model Registry and creates a new STAC Item in the STAC Catalog to represent the trained model artifact, linking it back to the MLflow run and the source data.



- **Inference Workflow:** This workflow executes a model to generate predictions for a user.
    1. An End-User interacts with a Frontend Application. They select a model from the catalog (populated by the STAC API) and specify an area of interest for analysis.
    2. The frontend sends an inference request to the Inference API Gateway.
    3. The gateway routes the request to Seldon Core, the model serving engine. The request specifies the model name and version to use.
    4. Seldon Core loads the appropriate model. It pulls the model's artifacts from the MLflow Model Registry and its associated object storage location in MinIO. The model is served within a containerized environment.
    5. The input data (e.g., a satellite image chip) is passed to the model, which generates a raw output (e.g., a raster mask).
    6. This raw output is then passed to a Custom Post-Processing Transformer, which is part of the Seldon Core inference graph. This transformer converts the raster mask into a vector-based GeoJSON format. (this is where postprocessing needs to be figured out , how to read post processing script for different kind of model as output would be different , more research is needed)
    7. The final GeoJSON result is sent back through the API gateway to the frontend, where it is visualized for the user (e.g., as points/lines/polygons overlaid on a map).

3. Technology Stack

The selection of each technology in this platform is a deliberate choice aimed at building a robust, scalable, and open-source MLOps ecosystem. This section provides a detailed analysis of each component, justifying its role within the architecture.

Table 1: Core Technology Stack Summary

Component	Role	Tool	Justification
Model & Data Catalog	Central registry for discovery and metadata.	STAC w/ MLM Extension	Open, community-driven standard for geospatial assets, perfectly suited for model discovery and linking models to the data they were trained on or can operate on.
Catalog Backend	Scalable, queryable database for STAC metadata.	PgSTAC (PostgreSQL + PostGIS)	Provides a robust, scalable, and high-performance backend with powerful spatial and temporal query capabilities essential for a geospatial catalog.
Experiment Tracking	Logging metrics, parameters, and artifacts for reproducibility.	MLflow Tracking	The de-facto industry standard for tracking ML experiments, ensuring every run is logged and reproducible. It is a core component in similar large-scale systems.
Model Packaging	Standardizing models for reproducible execution.	MLflow Projects & Models	A framework-agnostic format for packaging code, dependencies, and entry points, ensuring that a model can be run consistently anywhere.
Workflow	Executing	Argo	A lightweight, powerful,

Orchestration	multi-step, container-based pipelines.	Workflows	and Kubernetes-native workflow engine that excels at orchestrating container-based DAGs without the overhead of a full MLOps platform.
Model Serving	Deploying models as scalable, production-grade API endpoints.	Seldon Core	An advanced, Kubernetes-native serving framework with flexible inference graphs and support for custom transformers, crucial for post-processing steps.
Artifact Storage	Storing large files like model weights, logs, and datasets.	MinIO	A high-performance, S3-compatible, open-source object storage system that integrates seamlessly with both MLflow and DVC and can be self-hosted on Kubernetes.
Data Versioning	Versioning large-scale training and evaluation datasets.	DVC (Data Version Control)	A Git-like version control system for large data files, enabling reproducibility by linking specific data versions to code and model versions.

3.1 Catalog : STAC, MLM, and PgSTAC

The heart of the platform's discovery and standardization capability is the SpatioTemporal Asset Catalog (STAC). I myself love STAC so much because it is validated by the maturity of the STAC ecosystem and its extensions.

- **STAC as the "Card Catalog":** STAC is an open specification that provides a common language to describe geospatial information, making it easily searchable and discoverable. While traditionally used for satellite imagery and other Earth observation data, its flexible structure is perfectly suited to serve as the central "card catalog" for the platform's machine learning models. Each model will be represented as a STAC Item, allowing users and automated systems to query for models based on spatial, temporal, or thematic criteria.
- **MLM Extension :** The key to this approach is the STAC Machine Learning Model (MLM) Extension. This extension standardizes the description of ML models that operate on geospatial data. It provides a rich set of fields to capture all the metadata required to understand, discover, and execute a model, including:
    - mlm:name and mlm:architecture: The model's name and underlying architecture (e.g., "U-Net").
    - mlm:framework and mlm:framework_version: The ML framework used (e.g., "PyTorch", "1.13.1").
    - mlm:tasks: The specific task the model performs (e.g., "instance-segmentation").
    - mlm:input and mlm:output: Detailed descriptions of the model's expected input (e.g., required raster bands, normalization statistics) and the structure of its output.
    - **Asset Roles:** The extension defines specific roles for assets, such as mlm:model for the model weights, mlm:source_code for the training scripts, and crucially for this architecture, mlm:container to link to the model's Docker image.

- **PgSTAC :** A catalog based on static JSON files does not scale to thousands of models or millions of data items. To build a performant catalog, this architecture employs PgSTAC, which implements the STAC specification on a PostgreSQL database with the PostGIS extension. The platform will use **stac-fastapi-pgstac**, a ready-made FastAPI application, to expose the PgSTAC database as a fully compliant STAC API endpoint.This API will be the primary interface for the frontend application and internal services to interact with the catalog. I remember a saying from one of the pg conference , if you need a database why search for anything else when postgres is there ? haha ,Jokes apart postgresql is awesome

3.2 Experiment and Packaging Engine: MLflow

MLflow is the clear winner for managing the core ML lifecycle. Its adoption in other major geospatial AI platforms like AIOPEN and IBM's Geospatial Studio further validates this choice.The platform will leverage three of MLflow's key components.

- **MLflow Tracking:** This component provides a centralized server for logging and querying experiment data. Every training run executed by the platform will use the MLflow Tracking API to log parameters (e.g., learning rate, batch size), metrics (e.g., loss, accuracy, IoU), and artifacts (e.g., visualizations, model checkpoints).
- **MLflow Projects:** The MLproject file format is the beauty of the platform's model standardization strategy.It is a YAML file that defines a project's execution environment (via docker image) and its entry points (e.g., preprocess, train, postprocess, evaluate). By requiring every model to be packaged as an MLflow Project, we can ensure that the orchestration engine (Argo Workflows) has a consistent, reproducible way to execute the training code for any model, regardless of its underlying implementation (this could be anything) .
- **MLflow Model Registry:** The Model Registry acts as a central repository for managing the lifecycle of trained models. When a training pipeline succeeds, the resulting model is registered here. The registry provides crucial governance features, including model versioning (e.g., building-detector-v1, building-detector-v2) and lifecycle staging (e.g., Staging, Production, Archived). The serving gateway (Seldon Core) will pull models directly from this registry for deployment, because we need to make sure that only validated and approved model versions are served.

MLflow will be configured to use the platform's PostgreSQL instance as its backend store for tracking metadata and the MinIO instance as its artifact root for storing model files .

3.3 Orchestration Layer

The platform requires a workflow orchestration engine to automate the multi-step processes of training and batch inference. The primary candidates in the Kubernetes ecosystem are Kubeflow Pipelines (KFP) and Argo Workflows.

- **Kubeflow Pipelines (KFP):** KFP is a core component of the larger Kubeflow project, an end-to-end MLOps platform originating from Google. It provides a Python SDK for defining pipelines, a UI for visualizing runs, and its own metadata backend. It is tightly integrated with other Kubeflow components like Katib for

hyperparameter tuning and KServe for model serving.

- **Argo Workflows:** Argo is a CNCF (Cloud Native Computing Foundation) graduated project designed as a general-purpose, container-native workflow engine for Kubernetes. Each step in an Argo workflow is a container. It is lightweight, highly flexible, and does not bundle other MLOps components, focusing solely on workflow orchestration.

Table 2: Comparative Analysis of Workflow Engines

Feature	Kubeflow Pipelines	Argo Workflows	Analysis for this Platform
Scope	End-to-end MLOps platform	Dedicated workflow engine	Argo's focused scope is an advantage. It avoids redundancy with MLflow and Seldon, leading to a more modular, less complex "best-of-breed" stack.
Complexity	High. Requires installing and managing a large, interconnected set of components. Known for a steep learning curve.	Low to Medium. Lightweight installation as a set of Kubernetes CRDs. Easier to manage and understand.	The humanitarian goal of accessibility and maintainability strongly favors the lower complexity of Argo.
Flexibility	Opinionated towards its own ecosystem. Defining pipelines can be less flexible than raw YAML.	Highly flexible. Can orchestrate any container-based task. Workflows can be defined in YAML or via a Python SDK (Hera).	Argo's flexibility is ideal for executing the diverse, containerized models this platform will host.

Integration	Tightly integrated with KServe, Katib. Can be complex to integrate with external tools like MLflow.	Integrates easily with any tool that can be containerized or has an API. Perfect for an MLflow-centric architecture.	Argo's "pluggable" nature makes it the superior choice for integrating with our pre-selected components.
Community	Strong, backed by Google and a large community.	Very strong, CNCF graduated project with wide adoption in both MLOps and general CI/CD.	Both have strong communities, but Argo's focus on the core workflow problem makes its community support more relevant to our needs.

The analysis reveals that while Kubeflow is a powerful platform, its monolithic, all-in-one nature creates significant drawbacks for this project. Integrating mlflow and seldon core into a full Kubeflow installation would lead to redundant components (e.g., two metadata stores, two UIs) and unnecessary complexity, which is a frequently cited challenge of Kubeflow.

Argo Workflows, in contrast, is the ideal choice. It is a pure orchestrator. It does one thing and does it exceptionally well: run sequences of containers on Kubernetes. This aligns perfectly with the architectural decision to package every model as a container with a standard

MLflow Project interface. Argo will simply be the engine that executes these containers according to a defined workflow, without imposing any other part of an MLOps stack. This modular approach leads to a simpler, more maintainable, and more flexible system.

Final Decision: The platform will use **Argo Workflows** as its orchestration engine.

3.4 Model Serving Gateway

For production model serving, a simple Flask-based server like the one included in MLflow's default serving command is insufficient. It lacks scalability, advanced deployment strategies, and the ability to handle complex inference logic. The

architecture requires a robust, Kubernetes-native serving framework. The two leading open-source options are Seldon Core and KServe.

- **KServe:** Originally KFServing, KServe is the serving component of the Kubeflow ecosystem, now a standalone project. It excels at serverless deployment, offering features like scale-to-zero out of the box. Its primary abstraction is the InferenceService, which can be composed of a Predictor and an optional Transformer for pre and post-processing.
- **Seldon Core:** Seldon Core is a mature and highly flexible model serving framework. Its key feature is the concept of a declarative SeldonDeployment, which defines a complex inference graph. These graphs can include models, routers, combiners, and custom transformers, allowing for sophisticated deployment patterns like A/B testing, canary rollouts, and multi-armed bandits.

Table 3: Comparative Analysis of Serving Frameworks

Feature	KServe	Seldon Core	Analysis for this Platform
Inference Pipelines	Supports pre/post-processing via a Transformer component.	Supports highly flexible, multi-step Inference Graphs that can include custom Python components.	Both meet the critical requirement for a post-processing step. Seldon's graph is arguably more flexible for very complex, non-linear workflows.
MLflow Integration	Good. Can serve MLflow models, often via a Triton or MLServer backend.	Excellent. Provides a pre-built MLFLOW_SERVER implementation and has clear end-to-end tutorials for MLflow integration.	Seldon's direct and well-documented MLflow support is a significant advantage, reducing integration friction.

Local Development	Primarily designed for Kubernetes. Local testing can be complex.	Can be run with Docker Compose. This is a major advantage for local development and PoC stages.	Seldon's Docker Compose support is a perfect feature for this project's phased implementation plan. It allows developers to build and test the exact same inference graph locally before deploying to Kubernetes.
Community & Maturity	Actively maintained with a vibrant community, branched from Kubeflow.	Very mature project, commercially backed by Seldon, with a strong open-source community.	Both are mature and well-supported. The choice hinges on features, not community strength.
Advanced Features	Strong serverless capabilities (scale-to-zero).	Strong support for advanced deployment strategies (Canaries, A/B tests, etc.) and complex routing.	Both offer valuable advanced features.

The decision between Seldon Core and KServe is a close one, as both are excellent frameworks. However, for this specific project, Seldon Core holds a decisive advantage: the ability to run locally with Docker Compose. This feature is invaluable. It means that a data scientist can define and test a complete inference graph including the MLflow model and the custom raster-to-vector post-processing transformer on their local machine using the exact same SeldonDeployment manifest that will be used in production.

Final Decision: The platform will use **Seldon Core** as its serving gateway.

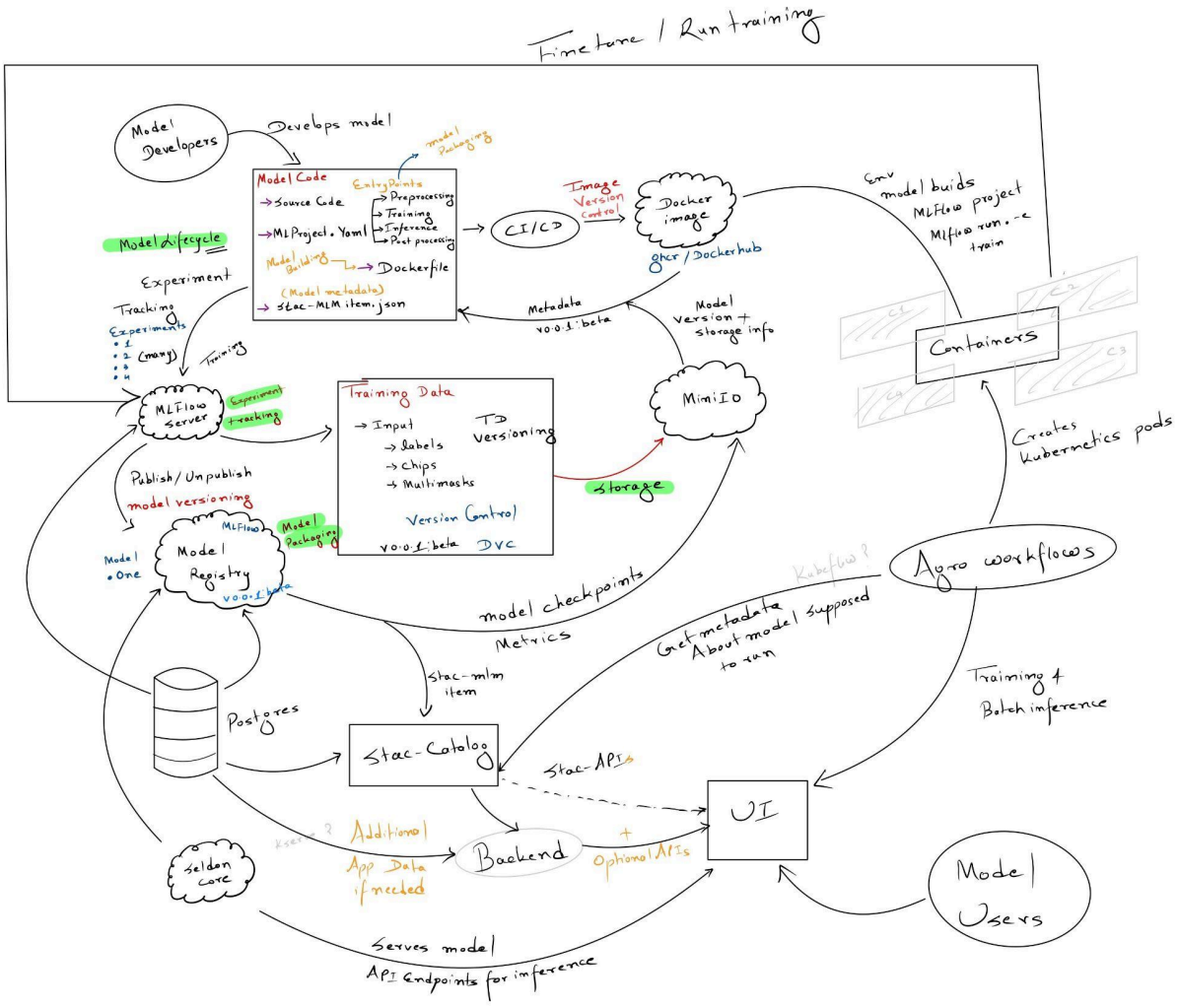
3.5 Storage : MinIO for Artifacts and DVC for Data Versioning

A scalable storage layer is the foundation of any MLOps platform. This architecture requires storage for two primary types of assets: MLflow artifacts (models, logs) and the large geospatial datasets used for training.

- **Object Storage with MinIO:** MLflow requires an S3-compatible object store to save large artifacts like model weight files and visualizations. MinIO is the industry-leading open-source, self-hostable object storage server. It is fully compatible with the S3 API, meaning any tool that works with AWS S3 will work with MinIO. It can be easily deployed on Kubernetes and configured as the artifact backend for MLflow, making it a natural fit for this open-source stack.
- **Versioning Large Geospatial Data:** A core principle of this platform is reproducibility, which requires versioning not only the code and model but also the exact dataset used for training. Storing large datasets (often many gigabytes of satellite imagery) directly in Git is infeasible, as Git is designed for text-based source code, not large binary files.
- **Data Version Control (DVC)** is the ideal solution to this problem. DVC is an open-source tool that works alongside Git to bring version control to data. The workflow is simple yet powerful:
    1. A user runs dvc add data/ to start tracking a folder of large files.
    2. DVC creates a small metadata file (e.g., data.dvc) that contains hashes and other information about the data. This small file is committed to Git.
    3. The actual data files are added to a DVC cache and are ignored by Git.
    4. When a user runs dvc push, the data from the cache is uploaded to a configured remote storage location (like an S3 bucket).
       This approach allows teams to version petabyte-scale datasets using Git workflows, without bloating the Git repository.

A key architectural simplification arises from the fact that both MLflow and DVC can use the same type of backend storage. Instead of deploying and managing separate storage systems, the platform will use a **single MinIO deployment** to serve both purposes. We can create distinct buckets within MinIO for example, mlflow-artifacts and dvc-data to keep the assets logically separated but managed by a single, unified storage service. This reduces infrastructure complexity and operational overhead, aligning with the principle of cost-effectiveness.

Figure 2 : Draft Ideation how things might work together


Timetone / Run training
Model Developers
Develops model
Model Code
Source Code
MLProject . Yaml
(Model metadata)
stac-MLM item.json
EntryPoints
Preprocessing
Training
Inference
Post processing
Dockerfile
CI/CD
Image Version Control
Docker image
ghcr / Dockerhub
Model lifecycle
Experiment
Tracking Experiments
Training
MLFlow server
Experiment tracking
Metadata v0.0.1-beta
Model version + storage info
Training Data
Input
Labels
Chips
Multimasks
TD Versioning
Version Control
v0.0.1-beta DVC
MinIO
Storage
Containers
Conv model builds MLFlow project
MLflow run . -e train
Creates kubernetics pods
Publish / Unpublish model versioning
Model One
MLFlow Model Registry
v0.0.1-beta
Model Packaging
model checkpoints
Metrics
Kubeflow ?
Argo workflows
Get metadata About model supposed to run
Training & Batch inference
stac-mlm item
Postgres
Stac-Catalog
Stac-APIs
UI
Ksonnet ?
Additional App Data if needed
Backend
+ Optional APIs
Seldon core
Model Users
Serves model API Endpoints for inference

---------- Draft ----------