

Recipe Suggester Tool

Operational Governance & Versioning Document

Baratto Luca, Moro Tommaso, Savorgnan Enrico

Contents

1	Version Control Strategy	2
1.1	Branching model	2
1.2	Versioning of code, data, and models	2
2	CI/CD Automation	4
2.1	Continuous Integration (CI)	4
2.2	Continuous Delivery & Deployment (CD)	4
3	Model Lifecycle Governance	6
3.1	Experiment Tracking & Training	6
3.2	Model Promotion Pipeline	6
3.3	Model Registry	6
3.4	Reproducibility	7
3.5	Experiment Tracking	7
4	Monitoring & Maintenance Plan	9
4.1	Infrastructure & Application Monitoring	9
4.2	Maintenance Procedures	9

1 Version Control Strategy

This section describes the VCS implemented for a controlled and tracked product deliverability. The main tool used by the team is GitHub, together with many related packages and extensions (Git LFS, GitHub Actions, GitHub Projects, GitHub Container Registry).

1.1 Branching model

The branching model follows common best-practices for fast product deployment.

- **main**: production-ready code.
- **project-manager**: integration branch for ongoing development.
- **feat/***: short-lived branches for new features implementation.
- **fix/***: short-lived branches for small fixes applied to **main**.
- **chore/***: short-lived branches for some general technicality.

The style of commits followed the same scheme, always derived from common best-practices:

- **feat: *short imperative***: used for new features implementation commits.
- **fix: *short imperative***: used for fixing pre-existing code.
- **refactor: *short imperative***: used for code refactoring.
- **chore: *short imperative***: used for updating grunt tasks.
- **docs: *short imperative***: used for documentation updates.

Pull Requests follows the same scheme.

1.2 Versioning of code, data, and models

The project enforces strict versioning across all three pillars of the ML application:

- **Code**: Versioned via **Git** on GitHub. The repository serves as the single source of truth for application logic, infrastructure config (**Dockerfile**), and pipeline definitions.
- **Data**: Versioned via **DVC (Data Version Control)**. Large datasets (images, labels) are stored remotely on **Microsoft Azure Blob Storage**, while lightweight **.dvc** pointers (containing MD5 hashes) are tracked in Git. This ensures that every code commit is linked to the exact dataset version used for training.

- **Models:** Versioned using a hybrid approach:
 - **Weights:** Final trained weights (`.pt`, `.onnx`) are tracked using **Git LFS** (Large File Storage) to keep the repository lightweight.
 - **Containers:** The inference environment is versioned as a Docker image in **GHCR**, tagged with the Git commit SHA (e.g., `recipe-suggester-models:sha-a1b2c3d`).

2 CI/CD Automation

The project adopts a robust Continuous Integration and Continuous Deployment pipeline to ensure code quality, automated testing, and seamless delivery. The pipeline is orchestrated via **GitHub Actions**, with **GitHub Container Registry (GHCN)** serving as the artifact store and **Railway** as the deployment platform.

2.1 Continuous Integration (CI)

The CI pipeline is defined in `.github/workflows/ci.yml` and triggers automatically on every Pull Request to the `main` branch. It ensures that no broken code merges into production by running parallelized checks for all three microservices.

The implemented CI jobs are:

1. **Models Change Detection:** A path-filter job that detects if changes occurred in the `code/models/` directory, avoiding unnecessary rebuilds of the heavy ML container.
2. **Models Quality Detection:** Executes a dedicated test suite (`yolo_test.py`) to validate the inference engine's robustness and accuracy. This includes:
 - **Edge Case Handling:** Verifies model behavior on *Black Images* (expecting no detections), *Empty Fridges*, and *Blurry Inputs* to prevent hallucinations.
 - **Sanity Checks:** Confirm that the model weights are loadable and that the `detect_ingredients` function returns valid predictions on standard sample images.
3. **Backend Verification:**
 - **Linting:** Uses `Ruff` to enforce Python code style and catch syntax errors.
 - **Testing:** Runs `pytest` with `pytest-cov` to execute unit tests and generate coverage reports, spinning up a temporary service container for PostgreSQL integration testing.
4. **Frontend Verification:**
 - **Type Check:** Runs `npx tsc --noEmit` to validate TypeScript static typing.
 - **Build Check:** Executes `npm run build` to ensure the React application compiles correctly for production.
5. **Docker Build Test:** Verifies that all `Dockerfiles` (backend, frontend, models) can be built successfully without errors, ensuring container validity before the CD phase.

2.2 Continuous Delivery & Deployment (CD)

The CD pipeline is defined in `.github/workflows/cd.yml` and triggers on every push to the `main` branch (i.e., after a PR merge).

- **Artifact Building:** The pipeline uses `docker/build-push-action` to build optimized multi-stage Docker images for the Frontend, Backend, and Models services.
- **Artifact Registry:** Images are tagged with both `latest` and the specific git-sha, then pushed to the **GitHub Container Registry (GHCR)**.
- **Deployment Strategy:** The deployment follows a “GitOps-like” pull model. Railway is configured to watch the GHCR registry. When a new image is pushed by the CD pipeline, Railway automatically detects the update, pulls the new image, and redeploys the service with zero downtime.

3 Model Lifecycle Governance

3.1 Experiment Tracking & Training

Experiments are conducted using the **Ultralytics YOLO** framework capabilities. To ensure traceability of every training run:

- **Configuration:** Training hyperparameters (epochs, batch size, learning rate) are defined in YAML configuration files (e.g., `code/models/yolo/config/config_yolo_ft.yaml`) which are committed to Git.
- **Artifacts:** Training outputs (confusion matrices, PR-curves, training loss logs) are generated in the local `runs/` directory. Key performance graphs are archived alongside the model weights to justify promotion.
- **Seeds:** A fixed random seed is enforced in the training script to ensure deterministic behavior and reproducibility of results across different runs.

3.2 Model Promotion Pipeline

The transition of a model from “Experiment” to “Production” follows a strict Gatekeeper process:

1. **Validation:** The candidate model is evaluated against the Test Set. It must meet the defined acceptance thresholds (e.g., $\text{mAP}@0.5 > 0.85$).
2. **Version Control:** The accepted model weights (`yolo_best.pt`) are renamed to include the version/date and committed to the repository via Git LFS.
3. **Integration:** The `code/models/app.py` service is updated to load the new weight file.
4. **CI/CD Trigger:** A Pull Request containing the new weights triggers the **Docker Build Test**. Upon merge, the CD pipeline builds a new Docker image containing the updated model and pushes it to GHCR.
5. **Deployment:** Railway pulls the new image, effectively promoting the model to the live production environment.

3.3 Model Registry

Instead of relying on a complex external model registry (like MLflow or AWS SageMaker), the project adopts a **Container-First Strategy**, utilizing **GitHub Container Registry** (GHCR) as the immutable source of truth for inference-ready models.

- **Immutable Artifacts:** Every model version is not just a weight file (`.pt`), but a full **Docker Image** (e.g., `recipe-suggester-models:sha-7b3f1a`) containing the weights, the inference code (`app.py`), and the exact system dependencies (`libgl1`, `torch`). This eliminates “it works on my machine” issues during deployment.
- **Deployment History as Registry:** The Railway platform serves as the deployment log. Rolling back to a previous model version is achieved by simply selecting a previous image tag from the deployment history, ensuring instant recovery without re-training or re-building.
- **Tagging Strategy:** Images are tagged with the specific **Git Commit SHA** in the CD pipeline. This establishes a bidirectional link: from a running container, one can trace back to the exact code and model weights version in the repository.

3.4 Reproducibility

Reproducibility is enforced through a “Configuration-as-Code” approach, ensuring that any model training run can be replicated by any team member.

- **Configuration Versioning:** Training hyperparameters (epochs, learning rate, batch size) are decoupled from the code and stored in versioned YAML files (e.g., `code/models/yolo/config/config_yolo_ft.yaml`).
- **Data Consistency:** The training dataset is tracked via **DVC (Data Version Control)**. The repository contains only the `data.dvc` pointer file (with MD5 hashes), ensuring that the code is always linked to the specific immutable snapshot of the dataset stored on Azure Blob Storage.
- **Environment Locking:** The execution environment is strictly defined via `requirements.txt` for Python packages and the `Dockerfile` for OS-level libraries (e.g., `libgl1`), preventing drift between training and inference environments.

3.5 Experiment Tracking

Since the project utilizes the **Ultralytics YOLO11** architecture, experiment tracking relies on structured local logging and Git-based promotion gates.

- **Native Metrics Logging:** For each training run, the system automatically logs key performance metrics (Box Loss, Class Loss, `mAP@0.5`, `mAP@0.5:0.95`) and generates visual artifacts (Confusion Matrices, Precision-Recall curves) in the local `runs/detect/` directory.
- **Manual Promotion Gate:** There is no automatic promotion. A “Human-in-the-loop” analyzes the generated artifacts. Only if the model surpasses the defined acceptance thresholds (e.g., `mAP@0.5 > 0.95` and inference latency `2s`), the Data Scientist manually commits the new `yolo_best.pt` to the repository via **Git LFS**.

- **Artifact Persistence:** While intermediate logs are ephemeral, the “winning” model’s performance graphs are archived in the repository documentation folder to provide a historical record of performance improvements over time.

4 Monitoring & Maintenance Plan

4.1 Infrastructure & Application Monitoring

Monitoring is implemented natively via the **Railway** observability suite, providing real-time insights into the health of the distributed system.

- **Resource Metrics:** CPU usage, RAM consumption, and Network bandwidth are tracked for every container (Frontend, Backend, Models, Postgres) to detect bottlenecks or memory leaks.
- **Application Logs:** `stdout` and `stderr` streams from all services are aggregated in the Railway console. This captures:
 - **Backend:** FastAPI access logs, validation errors, and interactions with external APIs (OpenAI).
 - **Models:** YOLO inference logs, including processing time and detection confidence scores.
 - **Frontend:** Nginx access logs and static asset delivery status.
- **Health Checks:** The Backend and Models services implement specific `/health` endpoints. Railway performs periodic probes on these endpoints; if a service becomes unresponsive, the platform automatically restarts the container to recover availability.

An important role is played by the (possible) Data drift, that can erase if the food presented in the image shared with the platform has a different distribution than the one seen during fine-tuning. However, no real-time automatic measures are currently been implemented to solve the issue, since for GDPR compliances the images are not stored. Only a “Human-in-the-loop” procedure can be used in such a situation, requiring manual analysis of the found ingredients, but without any ground-truth to test the drift.

In such a environment, the team decided to solve the issue directly at the root, while doing model fine-tuning, and opted for copy-past data augmentation for ensuring a uniform distribution of the ingredients, as discussed in System Specification Document at Section 4.2.

4.2 Maintenance Procedures

- **Database Backups:** Automated daily backups of the PostgreSQL volume are managed by Railway to prevent data loss.
- **Security Patching:** The multi-stage Docker builds utilize `python:3.11-slim` and `node:20-alpine` base images. These are updated automatically during the CI/CD build process whenever the pipeline runs, ensuring the latest security patches are applied to the OS layer.