# System Specification Document

S.O.L.A.R. Project

January 27, 2026

## Revision History

| Version | Date | Author | Description |
|---|---|---|---|
| 0.1 | 2025-12-17 | Matteo | Initial draft |
| 1.0 | 2026-01-26 | Matteo | Approved release |

# Contents

# 1 Version Control Strategy

This project uses Git for version control, with all source code, notebooks, data artifacts, and model artifacts maintained in a single GitHub repository. The workflow is designed to use small branches for feature development, with direct merges into the `main` branch upon completion.

## 1.1 Branching Model

A feature-based branching strategy is adopted.

- **main**: contains the stable and integrated state of the project. Only validated and complete features are merged into this branch.
- **<feature name>**: each developer works on their own feature branch (e.g., dashboard, API, DAO, model training, drift detection).

When a feature is completed and tested, it is merged directly into the `main` branch. This approach keeps the workflow simple and minimizes merge conflicts.

## 1.2 Code Versioning

Releases are identified using semantic versioning applied through Git tags. A version tag (e.g., `v1.0.0`) corresponds to a stable milestone, such as the delivery of the Minimum Viable Product (MVP) or the final system integration.

## 1.3 Data Versioning

Both raw and cleaned datasets are stored in the repository under dedicated directories. In a real case environment with larger datasets, a dedicated data versioning tool (e.g., DVC) would be recommended. Data preprocessing is deterministic and documented through scripts and notebooks, ensuring reproducibility. Since cleaned data is used to feed a synthetic sensor, data evolution is tracked implicitly through Git commits rather than a dedicated data versioning system.

## 1.4 Model Versioning

The project adopts a lightweight model versioning strategy consistent with the incremental learning paradigm used. Since the model is updated online and does not produce discrete training artifacts, no explicit model registry is employed.
Model behavior is versioned implicitly through:

- The Git commit history of the training and inference code.
- The configuration of the learning algorithm (e.g., Hoeffding Tree parameters and ADWIN settings).
- The monitoring metrics recorded during execution.

At any point, the active model state can be reproduced by re-running the system from the same code version on the same data stream. This approach ensures traceability and reproducibility while remaining compatible with a fully online learning workflow.

For ILSTM model, which is trained offline due to computational constraints, model artifacts are stored in a dedicated directory within the repository.

# 2 CI/CD & Automation

This project adopts a lightweight CI-oriented workflow. While a fully automated CI/CD pipeline is not implemented, several development and validation activities are partially automated and systematically enforced through version control practices.

### 2.1 Continuous Integration

Continuous Integration is achieved through disciplined use of Git and feature-based branching. Each feature is developed in an isolated branch and merged into the `main` branch only after appropriate local validation.

Depending on the scope of the change, the following activities are performed before merging:

- Execution of unit-level or functional tests when modifying core components such as API endpoints, the DAO interface, or dashboard logic.
- Manual validation of data consistency when changes affect preprocessing scripts or synthetic sensor generation.
- Verification that the Flask prediction service starts correctly and produces valid predictions when model-related code is updated.

This process ensures that the `main` branch always represents a stable and executable version of the system.

### 2.2 Automation of Model Training and Evaluation

Model training and evaluation are automated through dedicated scripts and notebooks. These artifacts allow:

- Reproducible training of regression models and incremental learners.
- Consistent evaluation using predefined metrics (e.g., Mean Absolute Error).

Although model retraining is not triggered automatically by a CI pipeline, the system supports controlled retraining through script execution when drift or performance degradation is detected.

### 2.3 Deployment Process

Deployment is performed manually. Once a stable version is reached, the Flask prediction service and Streamlit dashboard are launched in a controlled environment. Model updates are deployed by replacing the model artifact used by the prediction service, ensuring traceability through Git commits and version tags.

This approach prioritizes clarity, reproducibility, and governance over full automation.

## 3 Model Lifecycle Governance

The project adopts a lightweight model lifecycle governance strategy tailored for incremental learning and small-scale deployments. Models are managed using file-based versioning with metadata tracked in code and notebooks, ensuring traceability and reproducibility.

Each model script or artifact includes:

- Model type and architecture (Hoeffding Tree Regressor or ILSTM).
- Configuration parameters and hyperparameters.
- Features and data schema used for learning.
- Evaluation metrics (e.g., daily mean MAE).

Incremental models can be trained offline before deployment. They continuously learn from incoming streaming data provided by the DAO (synthetic sensor). Model updates are controlled using ADWIN, which detects statistically significant drift in prediction errors.

Governance actions triggered by ADWIN include:

- Increasing the learning weight of new samples to adapt quickly when drift is detected.
- Ignoring anomalous data points with very high prediction error to avoid degrading the model.
- Resetting or reinitializing the model if performance degrades persistently.

All model changes are tracked through Git commits, with parameters and configurations documented in code and notebooks, ensuring reproducibility and traceability of model evolution.

# 4 Monitoring and Maintenance

## 4.1 Monitoring

The system continuously monitors input features and model performance to ensure reliable predictions:

- **Feature Drift:** ADWIN monitors the stream of prediction errors for statistically significant changes.
- **Prediction Performance:** The custom KPI (daily mean MAE) is computed in real time at the end of the day, still using standard KPI to ensure the custom KPI is working as expected.
- **Alerts:** When ADWIN detects drift or the KPI exceeds predefined thresholds, alerts are triggered to notify operators.

## 4.2 Incident Response

When alerts are triggered due to detected drift or performance degradation, the influence of new samples is adjusted by modifying their learning weight:

- High weight for drift events to allow rapid adaptation.
- Zero weight for outlier points to prevent corruption.
- Standard weight for normal observations.

These procedures ensure the system maintains accurate, reliable predictions while providing a clear audit trail of operational decisions.