Room Occupancy Prediction

System Design and CI/CD Pipeline

Hamed Hemati, 22.10.2024

Development Tools and Platforms

- Code Versioning: GitHub
- CI/CD: GitHub Actions
- Containerization: Docker
- Docker Image Repository: DockerHub
- Container Management: Kubernetes
- Kubernetes Platform: Minikube (on a local machine)
- Monitoring: ElasticSearch + Kibana

Outline

0. Problem Definition

1. Model Training

2. Model Inference

3. CI/CD Pipeline for Deployment

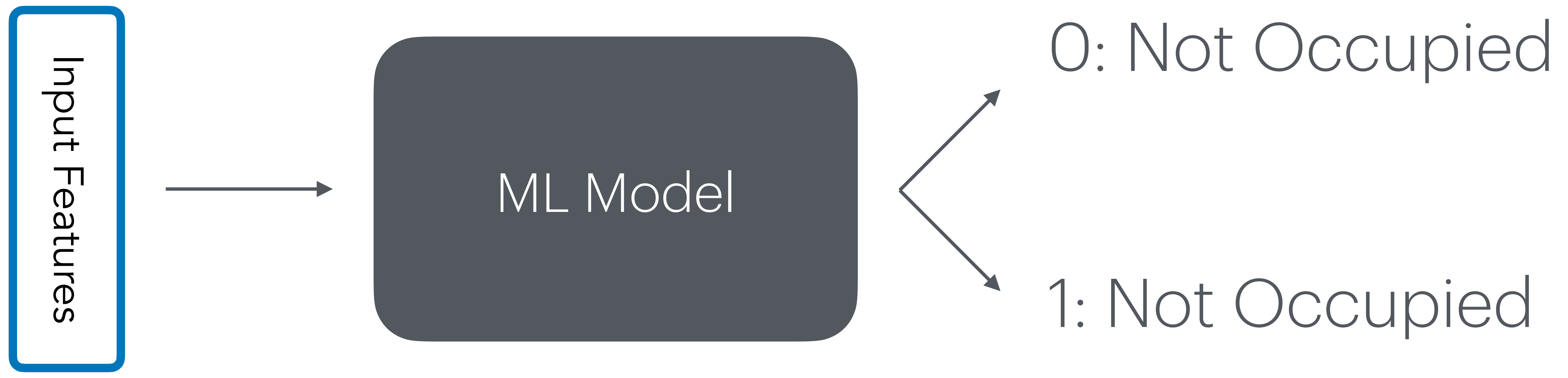
4. Scaling Deployment

5. Continuous Training

0. Problem Definition

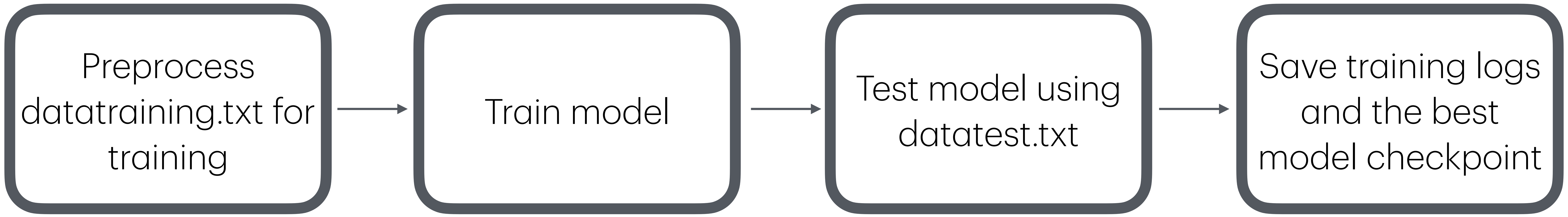
Objective

- **Goal:** Predict whether a room is occupied or not.
- **Input:** Set of features -> Temperature, Humidity, Light, CO2, Humidity Ratio
- **Output:** 0 (unoccupied) / 1 (occupied) -> Binary classification problem



Provided Data and Training Steps

- **datatraining.txt** -> Will be used for training and validation of the model.
- **datatest.txt** -> Will be used for testing the model's performance after training.
- **datatest2.txt** -> Will be used for monitoring the model's behavior "in production" to detect potential issues or drifts over time.


Preprocess datatraining.txt for training
Train model
Test model using datatest.txt
Save training logs and the best model checkpoint

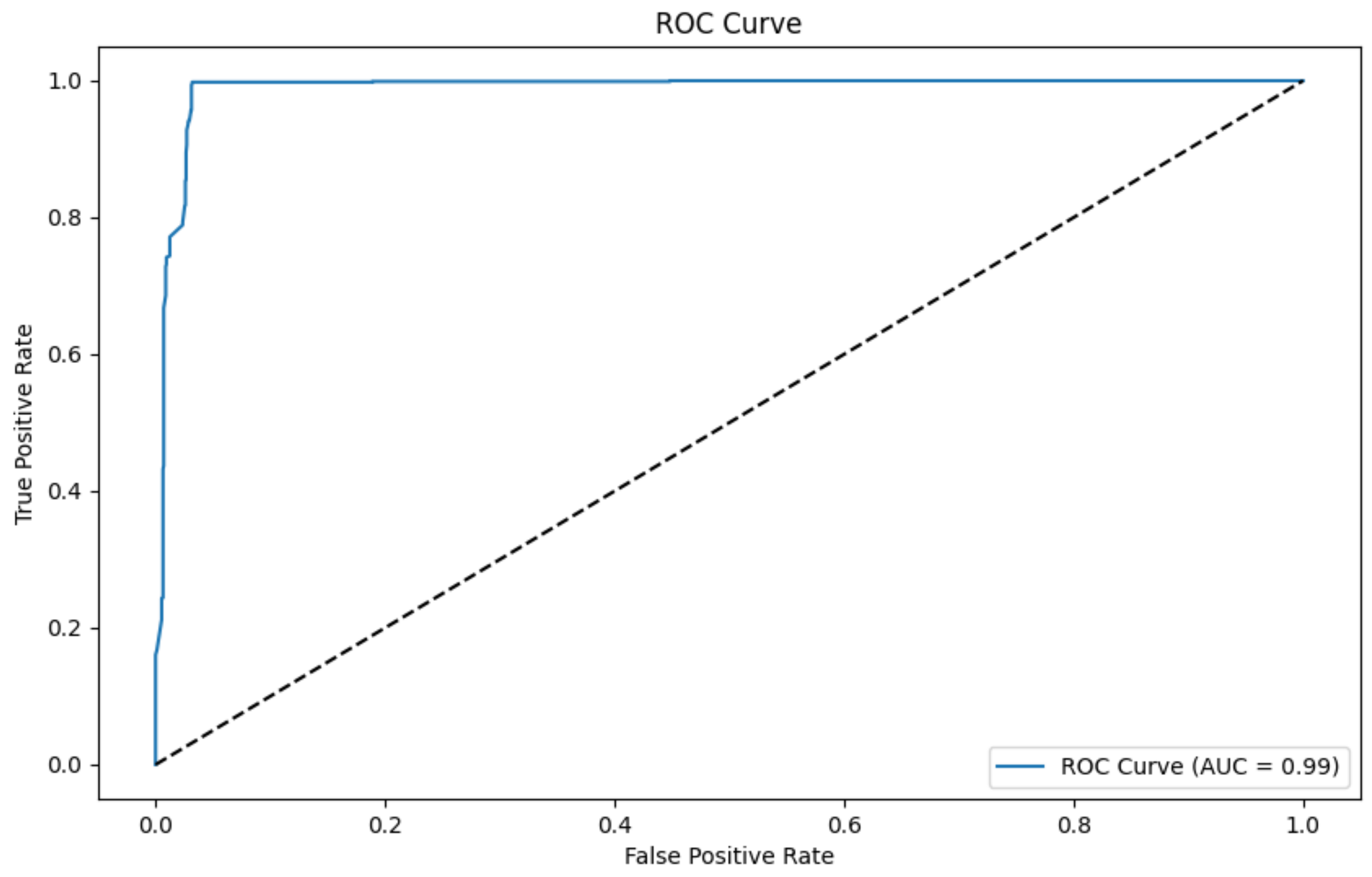
1. Training

Model Training

- **Model Used:** XGBoost.
- **Hyperparameter Tuning:** RandomizedSearch over a predefined search space for optimal performance.
- **Modular Design:** Training modules are implemented as a package to ensure reusability in testing and future tasks.
- **Extensibility:** With the modular design of the code, additional tools like the WandB/Tensorboard loggers can be easily integrated (though not the primary focus here).
- **Model Output:** The trained model is saved as a binary file for efficient inference.

Best Trained Model

- The best-trained model will be used for inference later.



Containerized Training

- Currently, three environment variables can be set for training on new datasets with arbitrary random seeds: **TRAINING_DATA**, **TEST_DATA**, and **SEED**. Additional variables can be added when needed.

- One method for providing arbitrary data is by mounting volumes and setting the corresponding environment variables.

```
docker  run  -v ./my_data:/src/my_data   -v ./out:/src/out                  \
             --env TRAINING_DATA="/src/my_data/datatraining.txt"            \
             --env TEST_DATA="/src/original/datatest.txt"                   \
             --env SAVE_PATH="/src/out" occupancy_training:latest
```

Additional Considerations for Model Training

- **Model Training as a Service:** The model training process can be exposed as an API, allowing external services or users to trigger training jobs programmatically. This could be useful in automated pipelines, where new data becomes available frequently, or in cases where users need to retrain models on-demand.

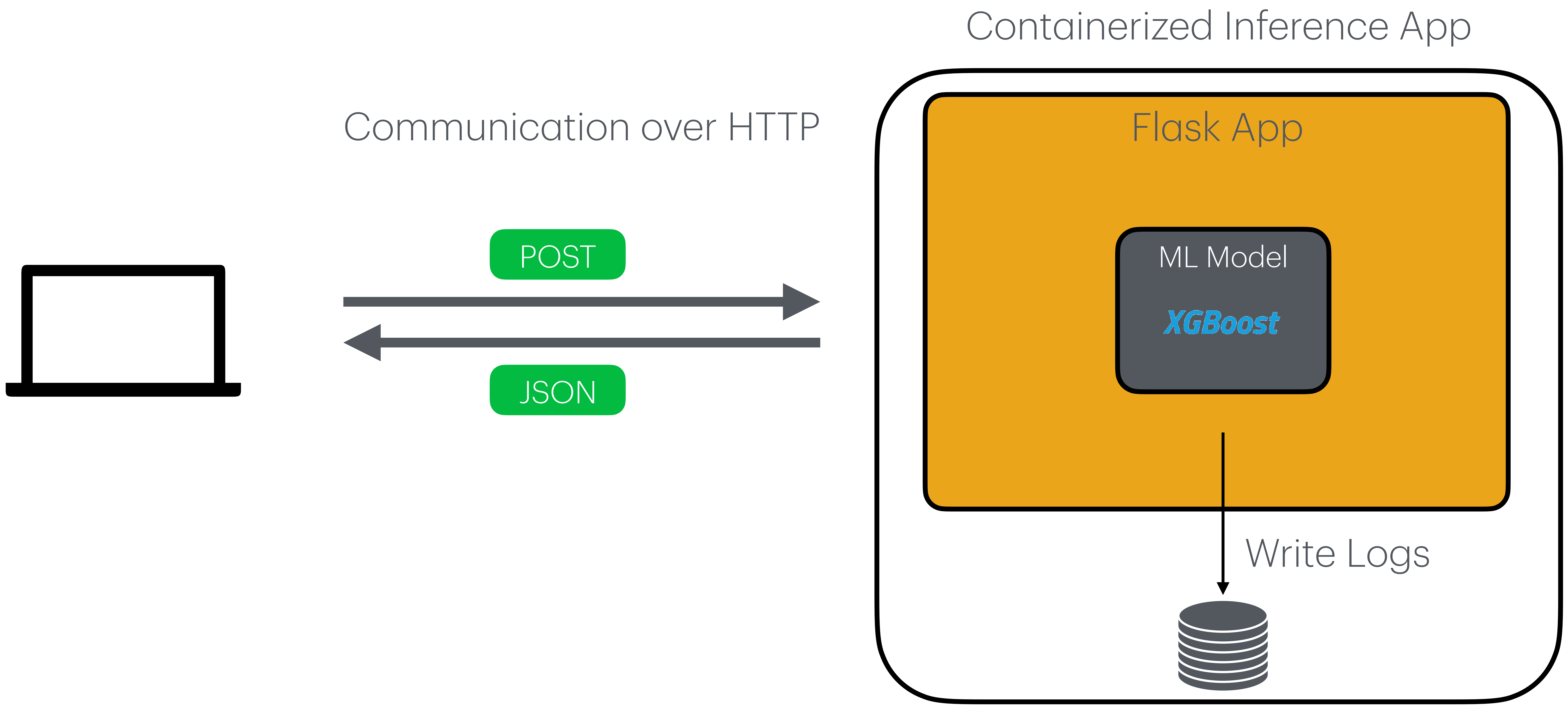
- **Modular Data Preprocessing:** A data preprocessing component can be integrated into the pipeline as needed. In this particular case, data preprocessing is not required, but the system is designed to allow easy addition of such a module, ensuring flexibility for future datasets or use cases that require transformation or cleaning steps.

2. Inference

Model Serving

- Model is served as an API for inference.
- Flask is used for serving the model.
- REST API endpoint (**/predict**) is exposed for predictions.
- Prediction logs are written in JSON format using pythonjsonlogger.
- A 0.1-second delay is added for testing and benchmarking.
- The model is containerized for fast deployment.

Model Serving: High-level Visualization



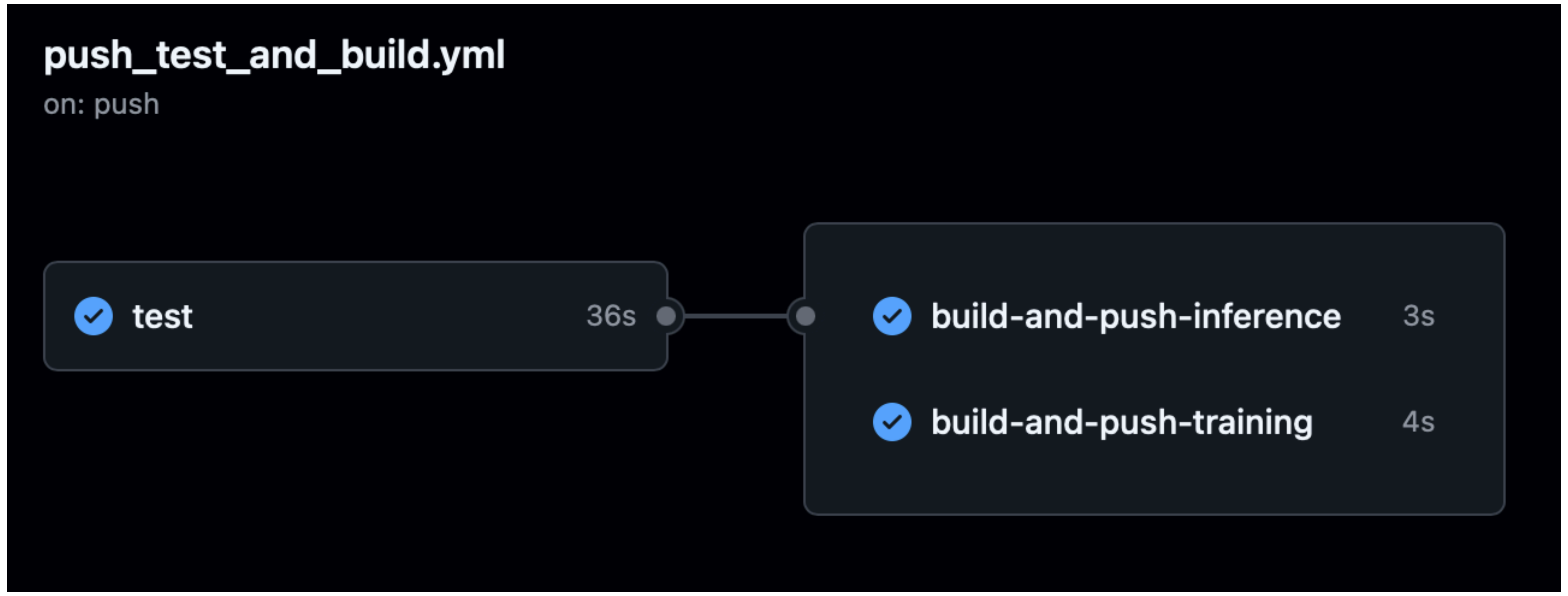
3. CI/CD

Unit Tests

- Unit tests ensure code changes do not break key functionalities.
- Three unit tests are currently implemented: **test_best_model_exists**, **test_load_model**, **test_model_accuracy**.
- New unit tests can be added as needed based on specific requirements.

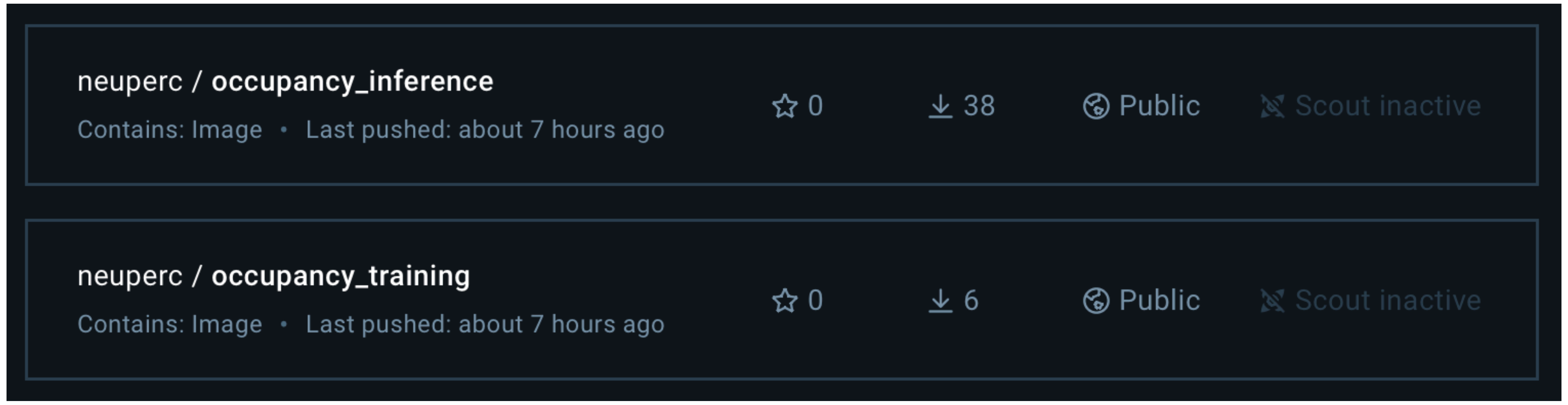
Workflow: Build and Push Container Images

- GitHub Actions is used to automate the CI/CD workflow.
- **Inference Workflow:** After running the tests, if changes are detected in the inference folder, unit tests are run, the inference container image is built and pushed.
- **Training Workflow:** After running the tests, if changes are detected in the training folder, training container image is built and pushed.



Docker Image Registry

- **neuperc/occupancy_inference:latest**: Docker image for model inference.
- **neuperc/occupancy_training:latest**: Docker image for model training.



neuperc / occupancy_inference
Contains: Image • Last pushed: about 7 hours ago
☆ 0 ⤓ 38 Public Scout inactive

neuperc / occupancy_training
Contains: Image • Last pushed: about 7 hours ago
☆ 0 ⤓ 6 Public Scout inactive

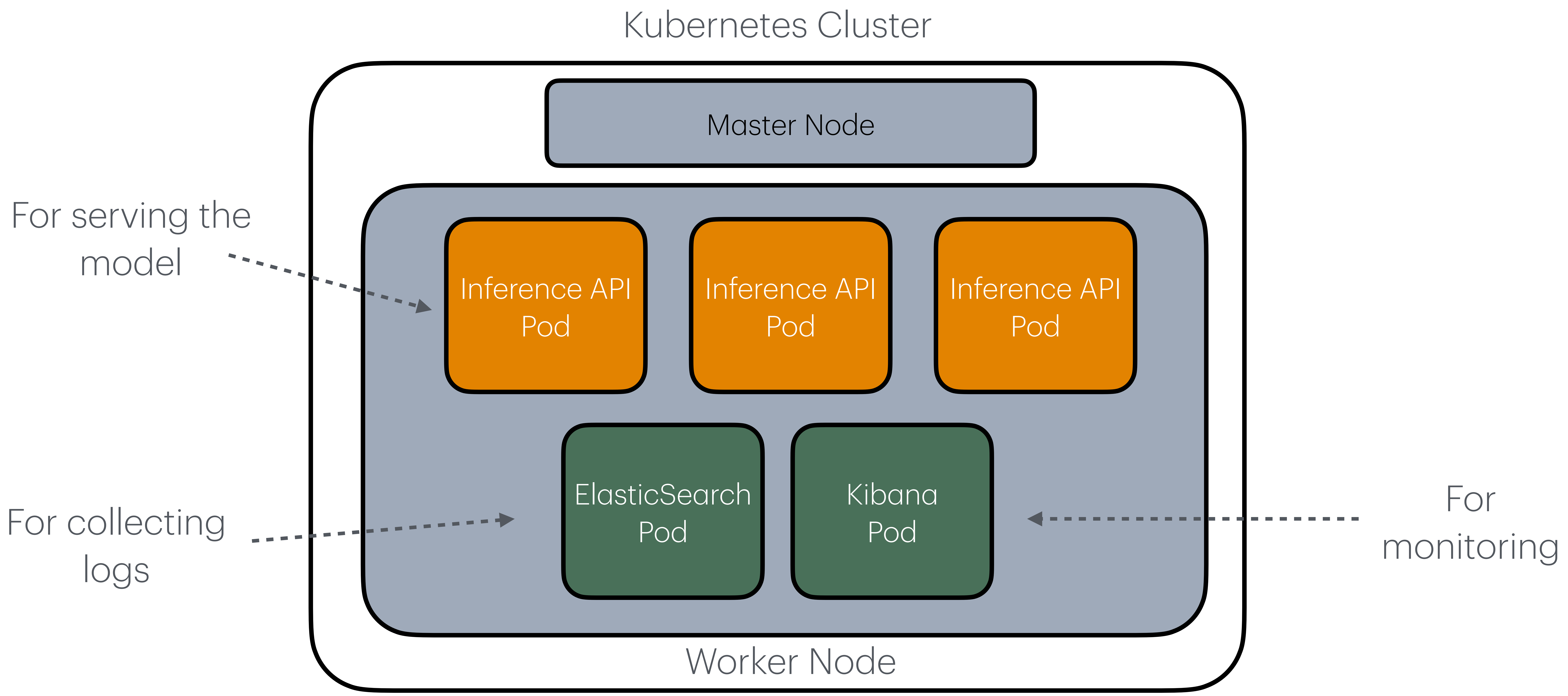
Pushed to Docker Hub

4. Scaling Up

Container Management and Orchestration

- High availability and automatic scaling are essential in production.
- Kubernetes is used to manage containerized training and inference workloads.
- Deployment files for inference and logging are available in the "**kubernetes/**" directory.
- Currently, there are two deployment files: one for the inference API and one for logging and monitoring.

Kubernetes Cluster Overview



Benchmarking

- 100 random requests are generated almost simultaneously to evaluate how the cluster handles high traffic and request load.

# Replicas	**1**	**2**	**3**
Duration	11.9 sec	6.2 sec	4.33 sec

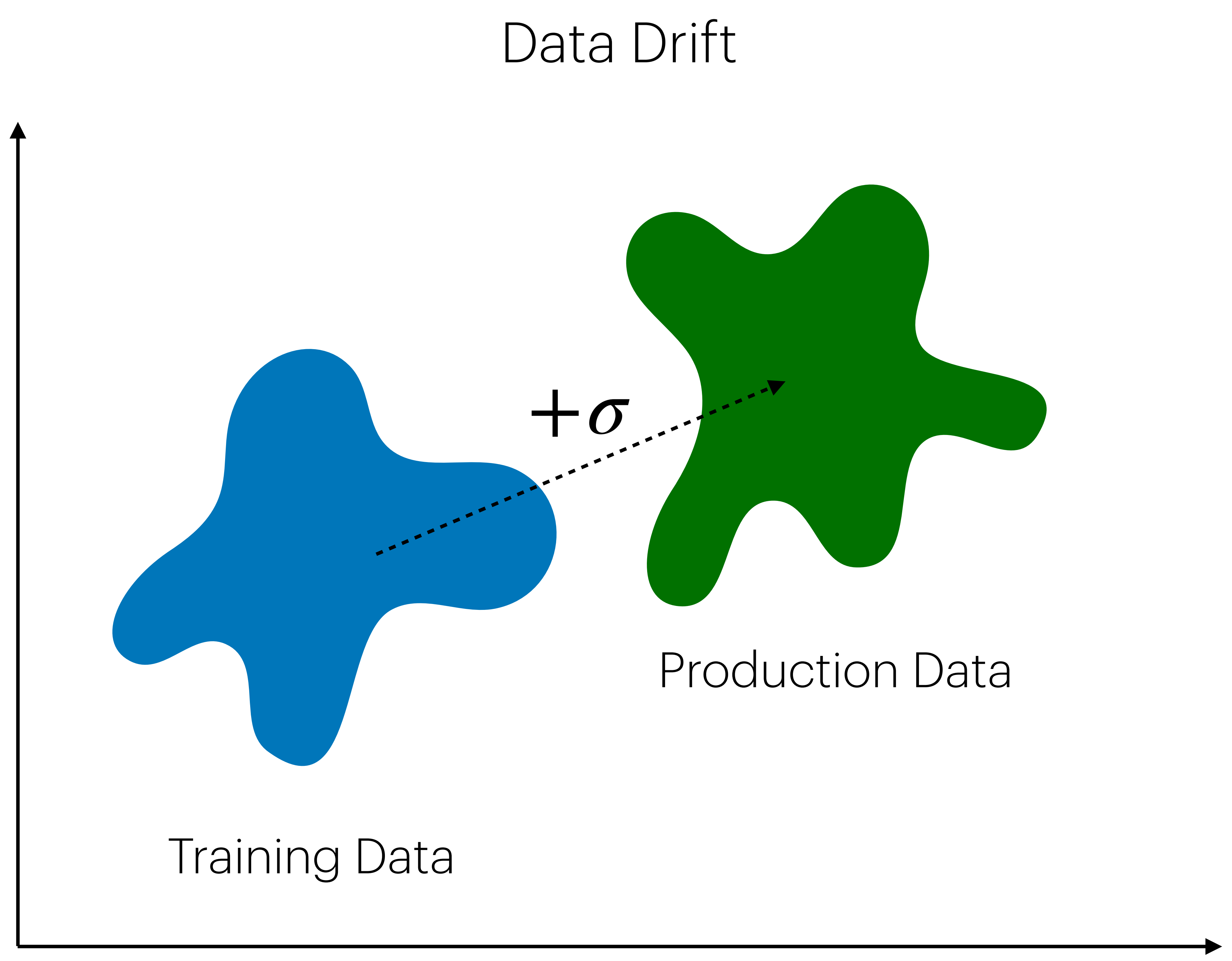
5. Continuous Training

Drift Detection

- Implementing a mechanism for detecting drifts is necessary for maintaining model performance.
- There are two main types of drift: **Concept Drift** (changes in input/ output mappings) and **Data Drift** (changes in the underlying data distribution).
- These drifts can lead to performance degradation and must be monitored continuously.
- Writing detailed and structured logs is essential for tracking model behavior and enabling effective drift detection.
- *Effective* drift detection is a critical step in the continuous training process to ensure the model adapts to new data over time.

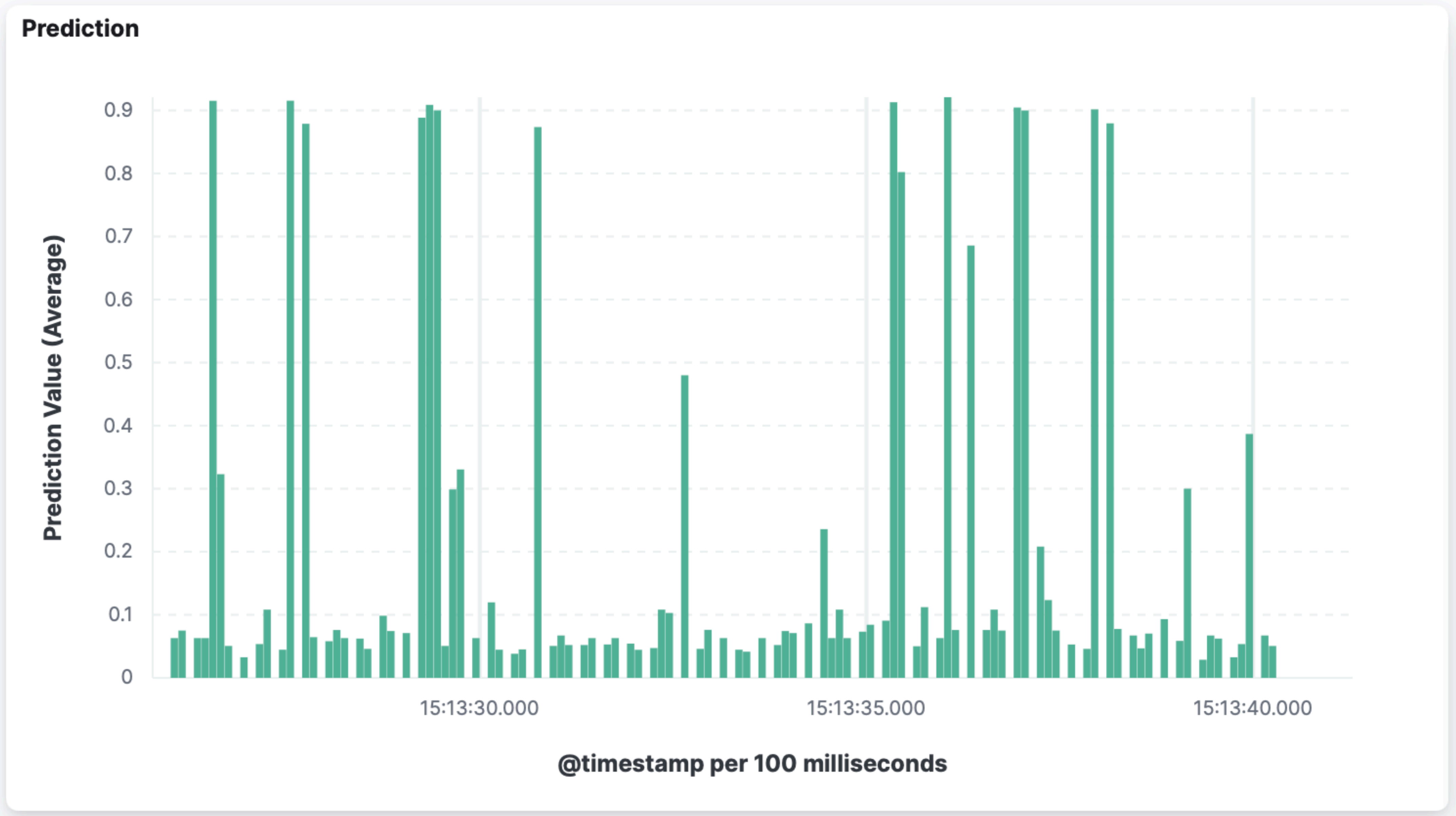
Monitoring: Logging Example

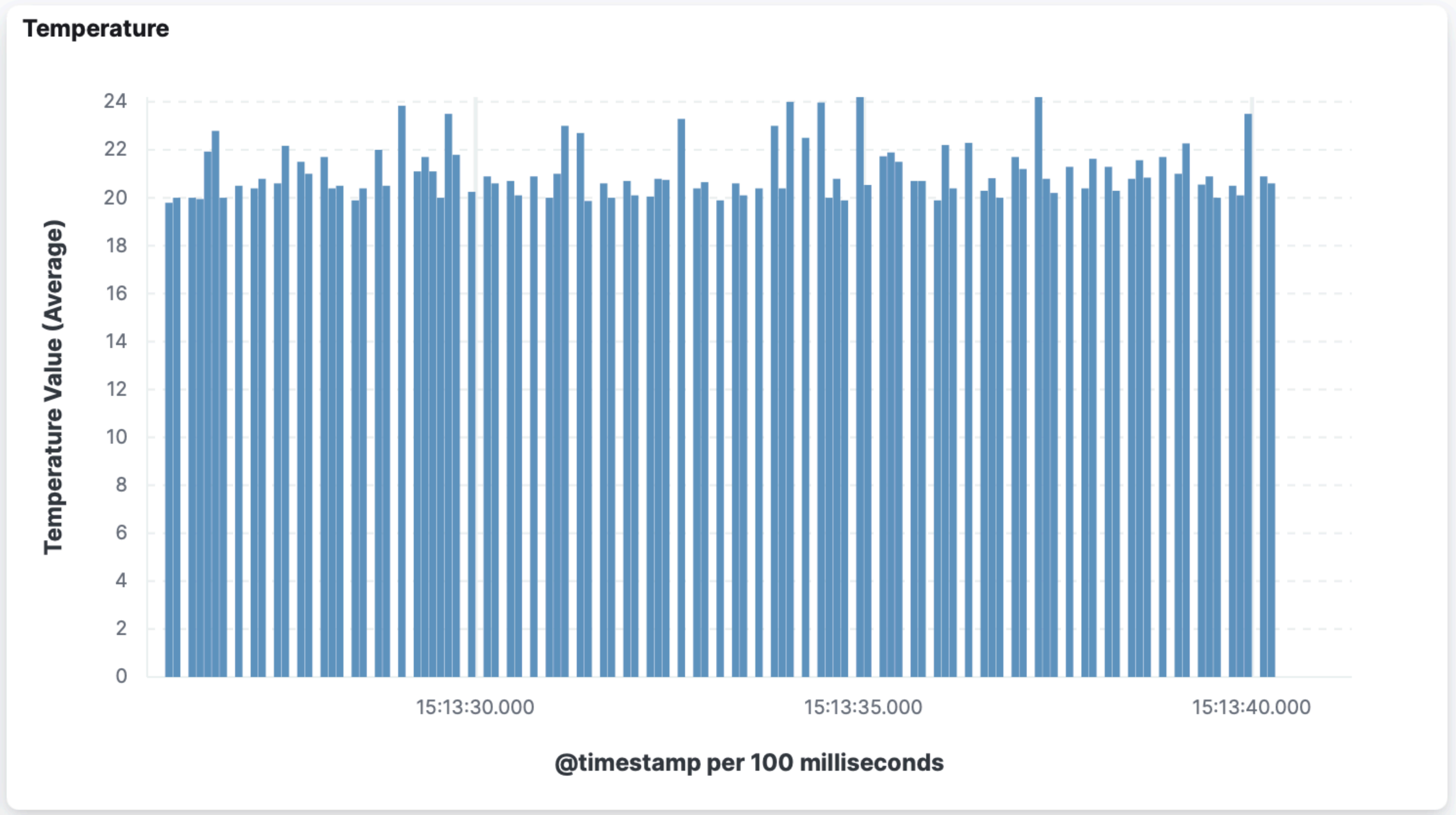
- Data drift can happen often in production environments.
- The choice of metrics and methods for drift detection depends on the specific problem and model being used.
- In our case, we can use two potential approaches for drift detection:
  1. Monitoring the model's prediction uncertainty.
  2. Monitoring data statistics, such as temperature (for simplicity, we only focus on this feature).



Monitoring: Normal Data Behavior

- 100 random requests generated using file *datatest2.txt*:



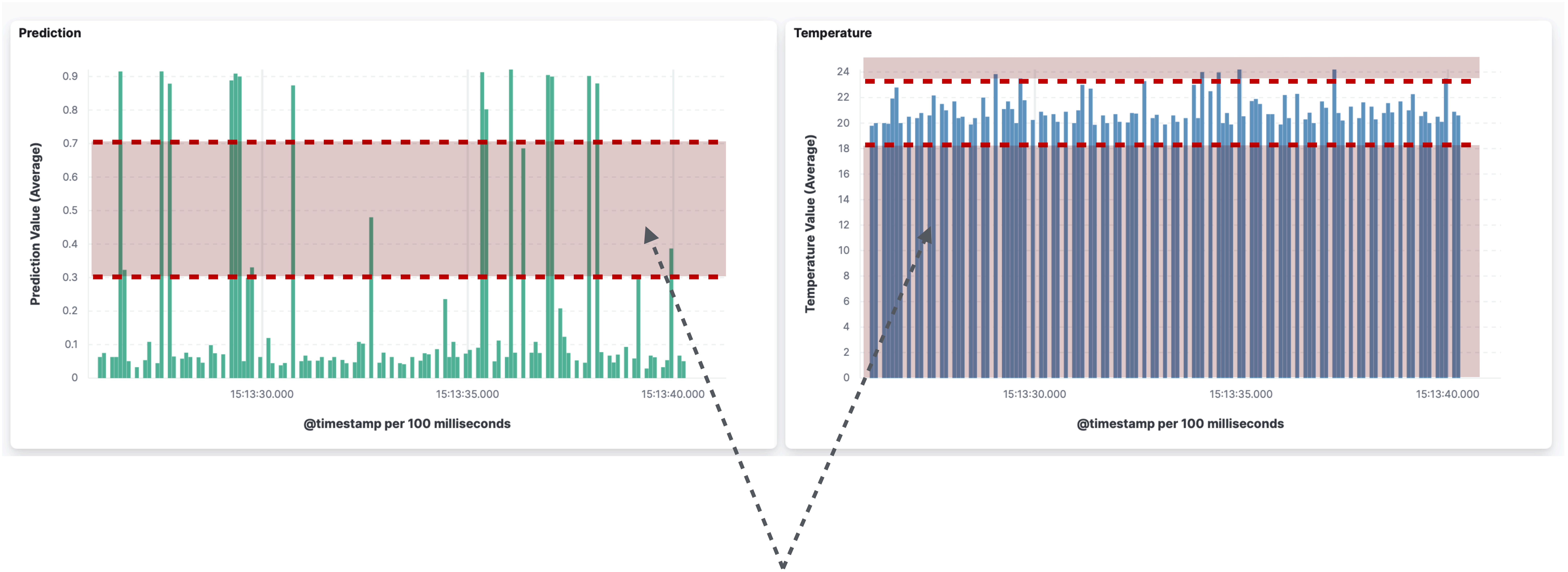


Logit values from model prediction
Normal range is either : [0.0, 0.3] or [0.7, 1.0]
Outside this range -> [0.3, 0.7] mean high uncertainty

Temperature values of the input features.
Expected value of temperature is: 20.0

Monitoring: Normal Data Behavior

- 100 random requests generated using file *datatest2.txt*:

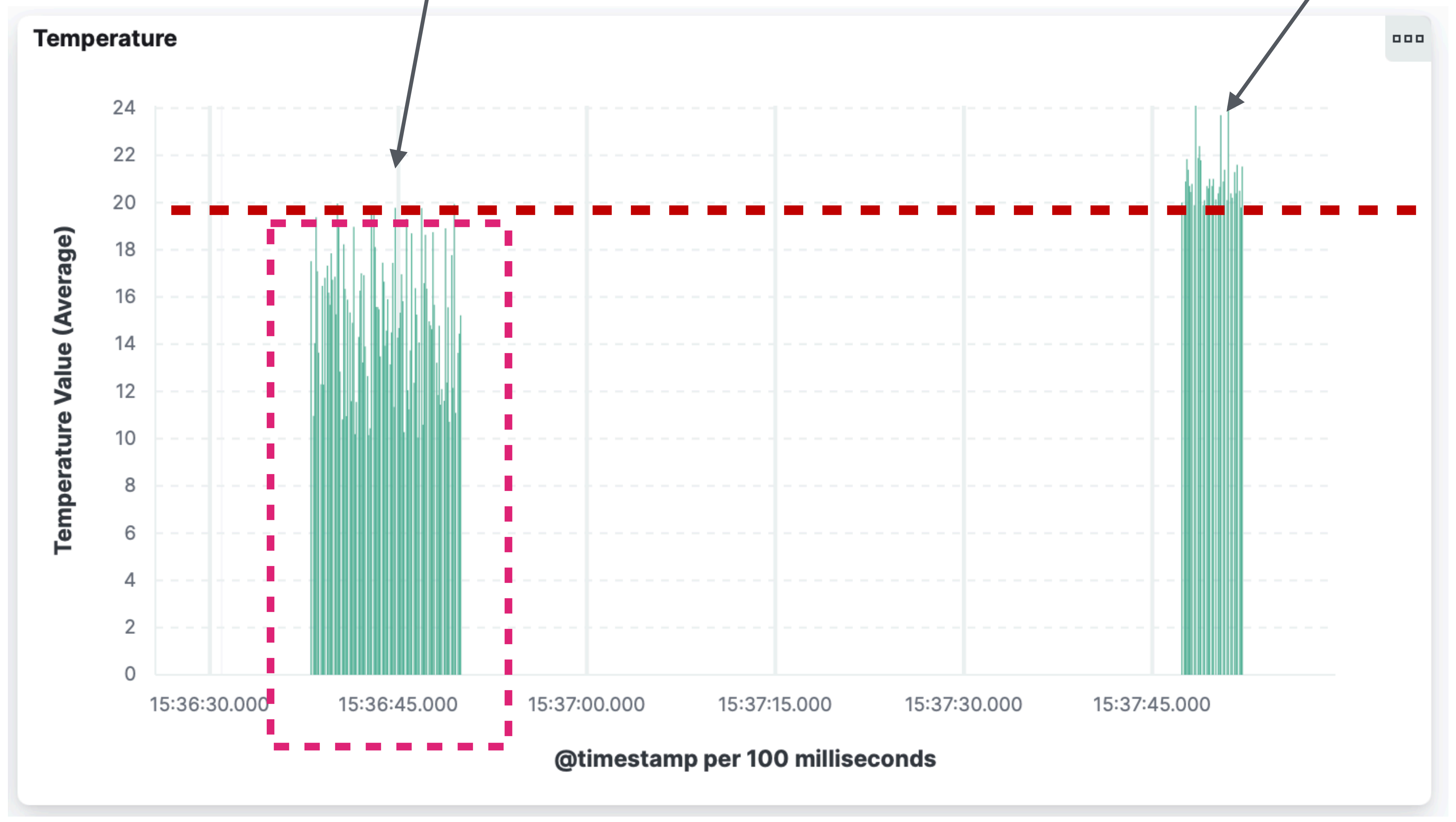


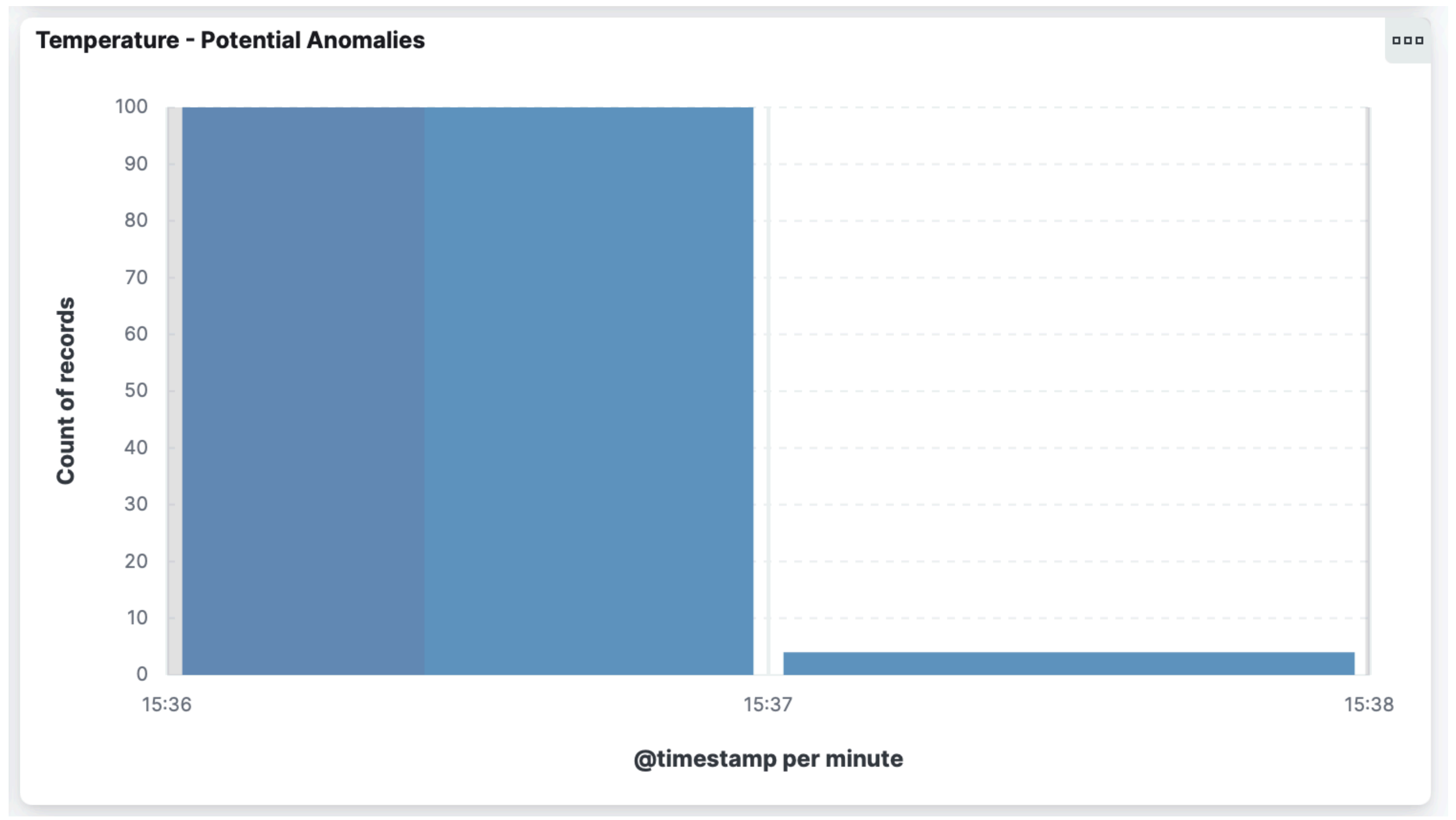
These ranges could mean a change in model/input behavior.

Example: Temperature Out of Range

Manually generated requests

Requests from datatest2.txt

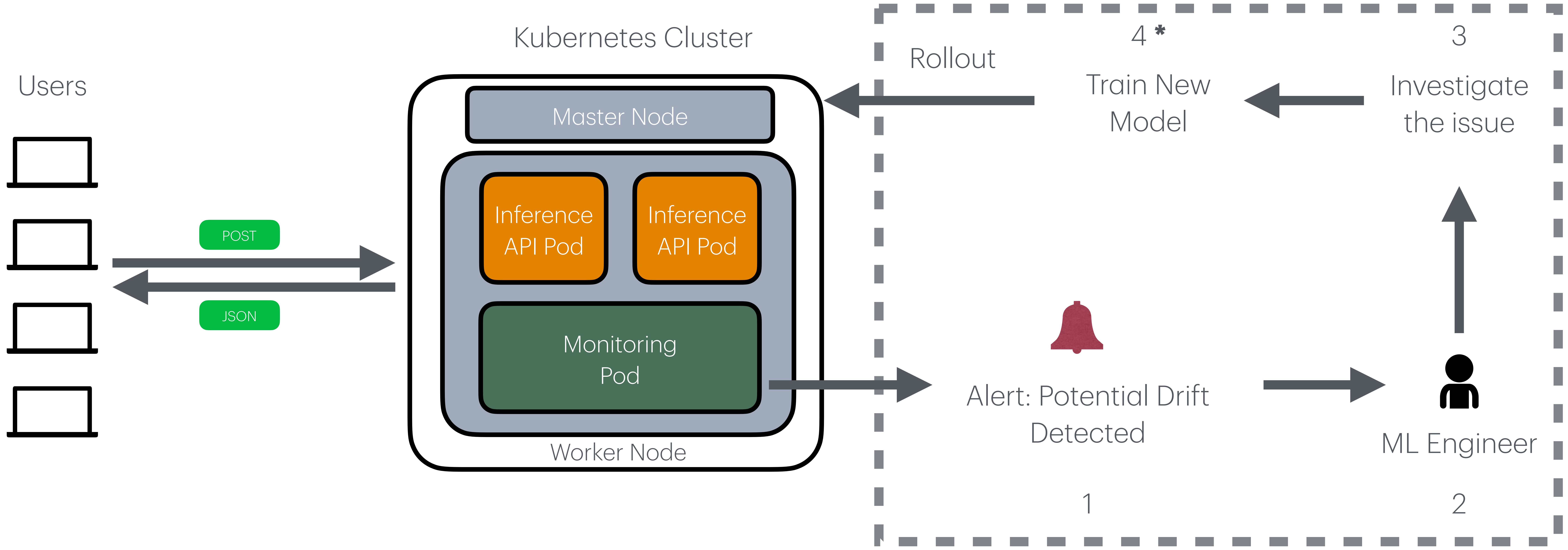




Temperature value in each request per milisecond

Number of "potential anomalies" per minute

Continuous Training: High Level Picture



*: New data may need to be provided for step 4.

Continuous Training: Full Automation

- Full automation of the continuous training pipeline can also be achievable.
- However, incorporating new data is often necessary to ensure model improvement.
- Example of a fully automated workflow:
  1. Potential drift is detected in the model's performance.
  2. Recent requests are collected and evaluated as new data points.
  3. A new model is trained using the same training set, with additional emphasis on reducing uncertainty in the recent data trends (possibly with adjusted hyperparameters).
- For many problems, collecting new (labeled) data is crucial for meaningful model improvements over time.

Other Relevant Approaches

- **Continual Learning:** Enhances training efficiency and allows the model to incrementally learn from the latest data trends without requiring a full retraining, and without significant degradation on past data. This method helps the model adapt over time to evolving data distributions.

- **AutoML:** Automates the entire machine learning pipeline, including model selection, hyperparameter tuning, architecture design and evaluation. It reduces manual intervention, and can enable faster and more efficient model development, especially for problems requiring frequent updates.

Demo