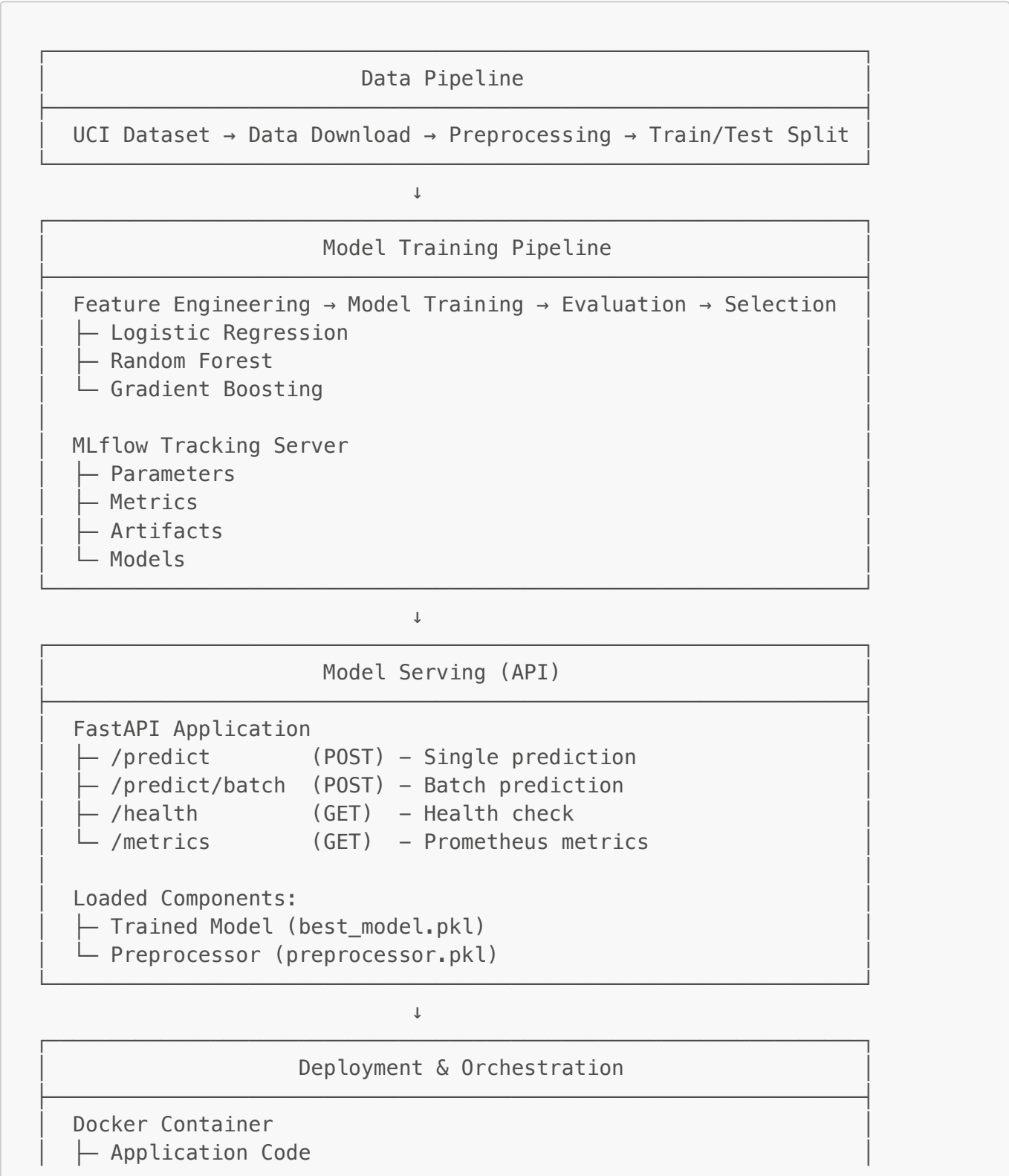


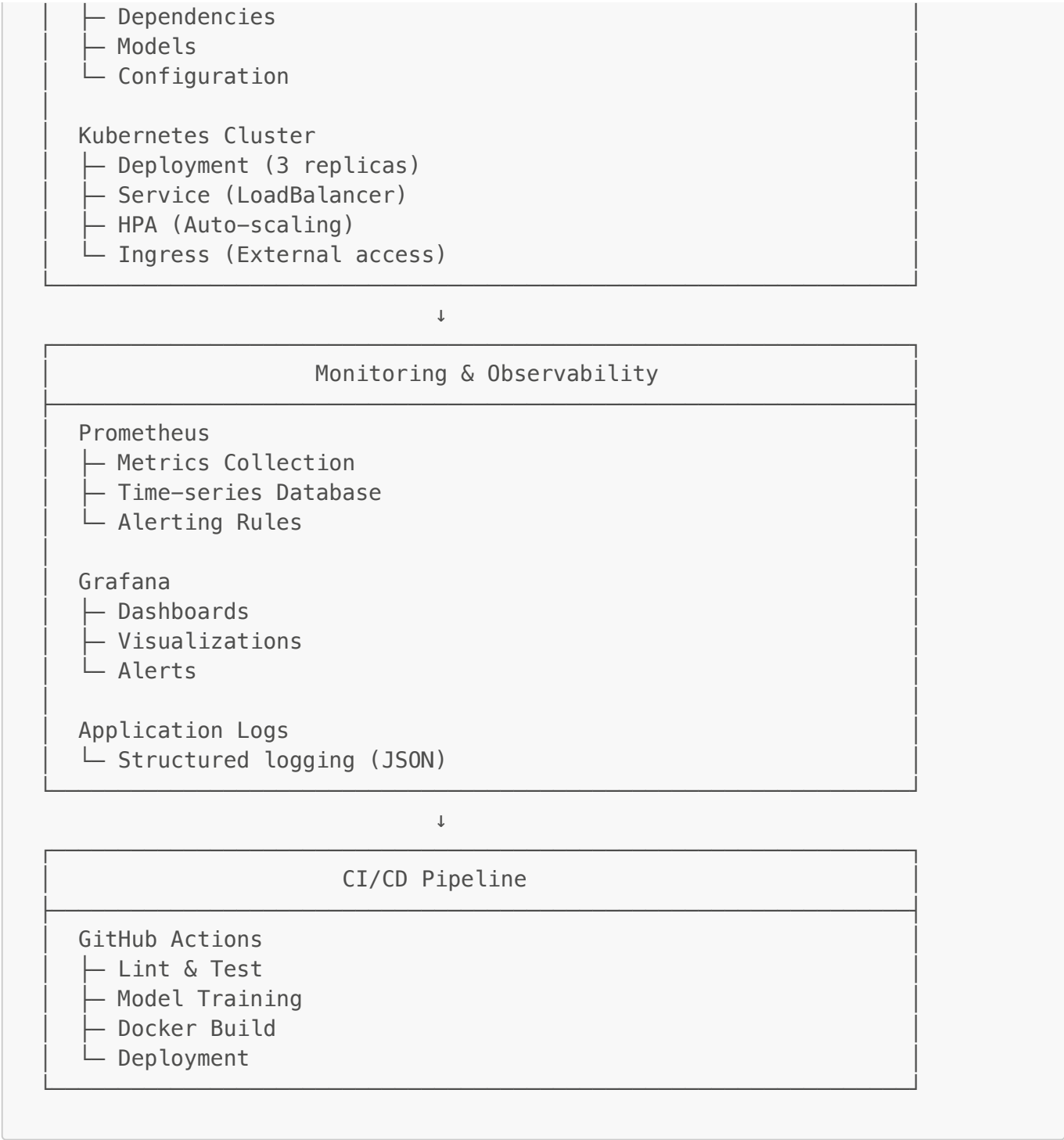
# System Architecture - Heart Disease Prediction MLOps

## Overview

This document describes the complete architecture of the Heart Disease Prediction MLOps system, including data flow, component interactions, and deployment architecture.

## 1. High-Level Architecture





## 2. Component Details

### 2.1 Data Pipeline

**Components:**

- `src/download_data.py`: Data acquisition and initial processing
- `src/preprocessing.py`: Feature engineering and preprocessing

**Data Flow:**

1. Download raw data from UCI repository
2. Handle missing values (median imputation)

3. Convert target to binary classification
4. Split into train/test sets (80/20)
5. Apply feature scaling (StandardScaler)
6. Save processed data

**Outputs:**

- `data/processed/heart_disease.csv`
- `models/preprocessor.pkl`

## 2.2 Model Training Pipeline

**Components:**

- `src/train.py`: Model training orchestration
- MLflow: Experiment tracking

**Models Trained:**

### 1. Logistic Regression

- Linear model baseline
- Fast training and inference
- Interpretable coefficients

### 2. Random Forest

- Ensemble method
- Feature importance
- Handles non-linear relationships

### 3. Gradient Boosting

- Advanced ensemble
- Best performance
- Sequential learning

**Tracked Metrics:**

- Accuracy (train and test)
- Precision, Recall, F1-Score
- ROC-AUC
- Cross-validation scores
- Confusion matrix
- ROC curve

**Artifacts:**

- Trained models
- Feature importance plots
- Confusion matrices
- ROC curves

2.3 API Service

**Framework:** FastAPI

**Endpoints:**

Endpoint	Method	Description
/	GET	API status
/health	GET	Health check
/predict	POST	Single prediction
/predict/batch	POST	Batch predictions
/metrics	GET	Prometheus metrics
/docs	GET	API documentation

**Input Schema:**

```
{
  "age": float,
  "sex": int (0/1),
  "cp": int (1-4),
  "trestbps": float,
  "chol": float,
  "fbs": int (0/1),
  "restecg": int (0-2),
  "thalach": float,
  "exang": int (0/1),
  "oldpeak": float,
  "slope": int (1-3),
  "ca": float (0-3),
  "thal": float (3/6/7)
}
```

**Output Schema:**

```
{
  "prediction": int (0/1),
  "probability": float (0-1),
  "risk_level": string ("Low"/"Medium"/"High"),
  "timestamp": string (ISO 8601)
}
```

**Features:**

- Input validation (Pydantic)

- Request logging
- Error handling
- CORS support
- Health checks
- Prometheus metrics

## 2.4 Containerization

### Docker Image:

- Base: `python:3.9-slim`
- Size: ~500MB
- Layers:
  1. System dependencies
  2. Python dependencies
  3. Application code
  4. Models

### Container Configuration:

- Port: 8000
- Health check: `/health` endpoint
- Environment: Production
- Restart policy: Unless stopped

## 2.5 Kubernetes Deployment

### Resources:

#### 1. Deployment

- Replicas: 3 (default)
- Max replicas: 10 (HPA)
- Resource requests:
  - CPU: 250m
  - Memory: 256Mi
- Resource limits:
  - CPU: 500m
  - Memory: 512Mi

#### 2. Service

- Type: LoadBalancer
- Port: 80 → 8000
- Protocol: TCP

#### 3. HorizontalPodAutoscaler

- Min replicas: 2
- Max replicas: 10
- Metrics:

- CPU: 70%
- Memory: 80%

#### 4. Ingress

- TLS enabled
- Domain: heart-disease-api.example.com
- Certificate: Let's Encrypt

## 2.6 Monitoring Stack

### Prometheus:

- Scrape interval: 10s
- Retention: 15 days
- Metrics:
  - `prediction_requests_total`: Counter
  - `prediction_latency_seconds`: Histogram
  - `predictions_by_class`: Counter

### Grafana:

- Port: 3000
- Dashboards:
  - Request rate
  - Latency percentiles
  - Error rate
  - Predictions distribution

### Alerting Rules:

- High error rate (>5%)
- High latency (>1s p99)
- Pod failures
- Resource exhaustion

## 2.7 CI/CD Pipeline

### GitHub Actions Workflow:

#### Stage 1: Lint & Test

- Checkout code
- Set up Python
- Install dependencies
- Run flake8, black
- Run pytest with coverage
- Upload coverage reports

#### Stage 2: Train Model

- Download data
- Train models
- Track with MLflow
- Archive artifacts

### Stage 3: Build Docker

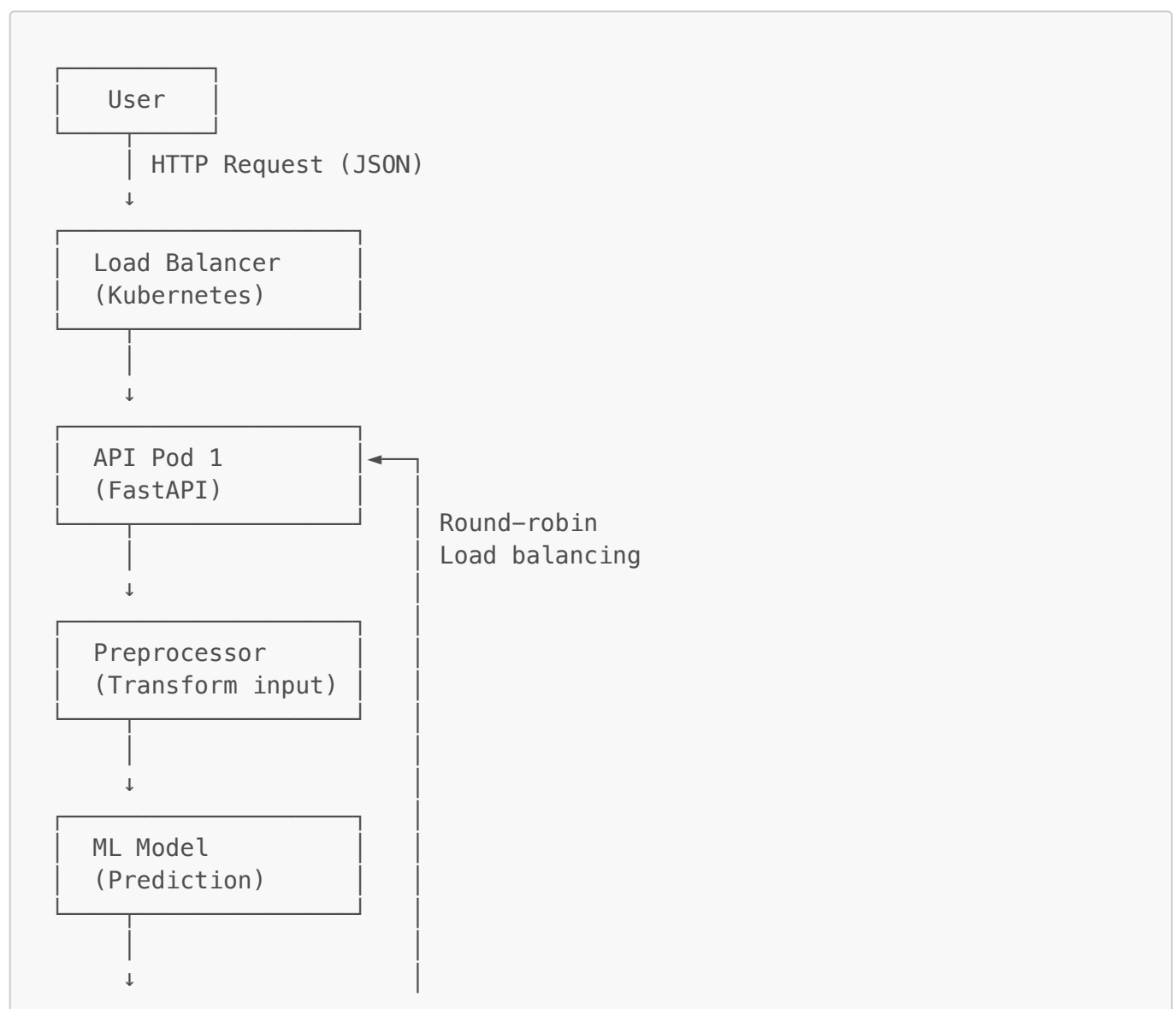
- Download model artifacts
- Build Docker image
- Test container
- Save image artifact

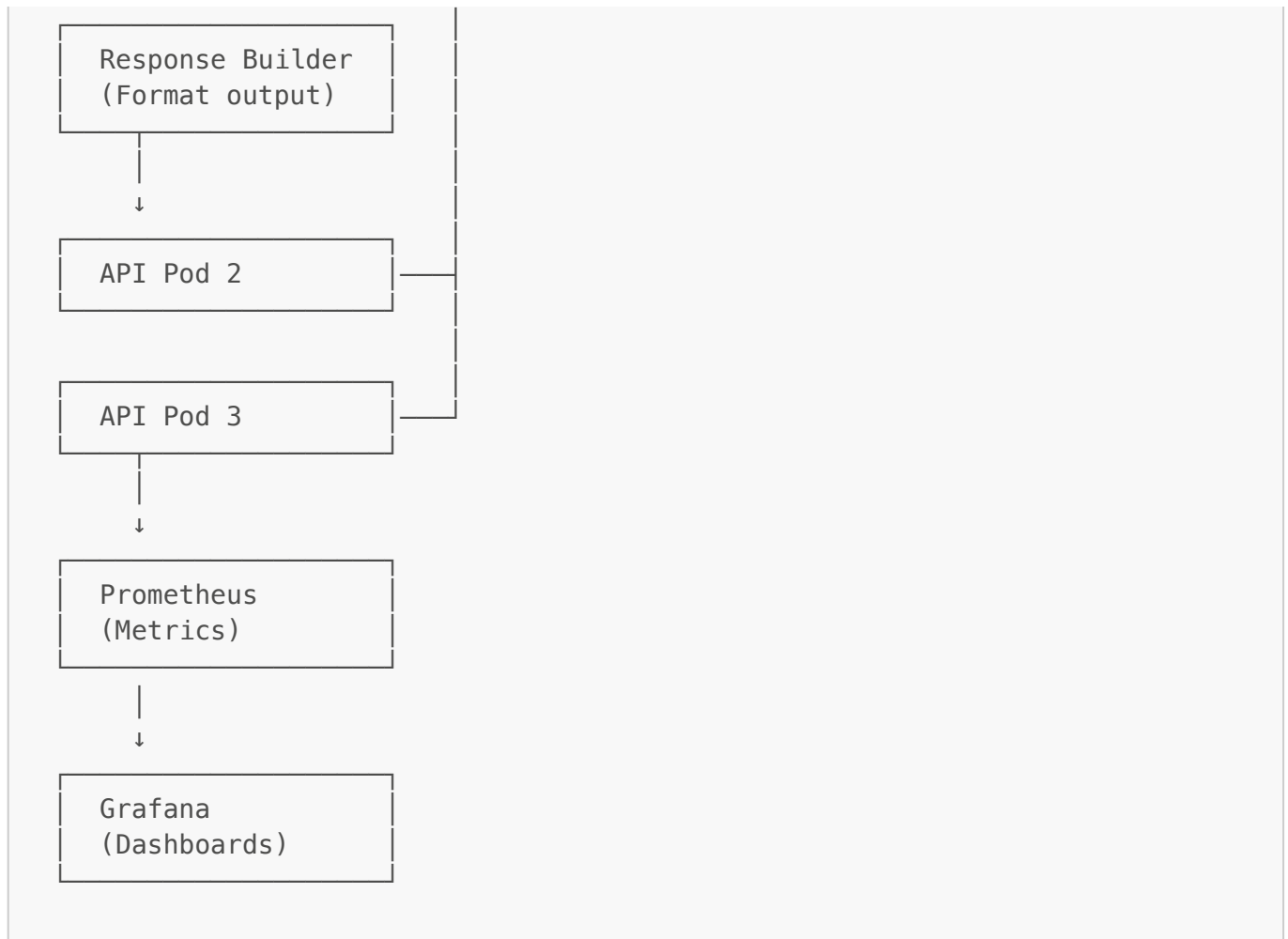
### Stage 4: Deploy

- Load Docker image
- Push to registry
- Deploy to Kubernetes
- Run smoke tests

---

## 3. Data Flow Diagram





---

## 4. Technology Stack

### Programming Languages

- Python 3.9+

### ML & Data Science

- scikit-learn: Model training
- pandas: Data manipulation
- numpy: Numerical operations
- matplotlib, seaborn, plotly: Visualization

### MLOps

- MLflow: Experiment tracking
- joblib: Model serialization

### API & Web

- FastAPI: API framework
- uvicorn: ASGI server
- pydantic: Data validation



## Testing

- pytest: Testing framework
- pytest-cov: Coverage reporting

## Code Quality

- black: Code formatting
- flake8: Linting
- pylint: Static analysis

## Containerization & Orchestration

- Docker: Containerization
- Kubernetes: Orchestration
- docker-compose: Local multi-container

## Monitoring

- Prometheus: Metrics collection
- Grafana: Visualization
- prometheus-client: Python library

## CI/CD

- GitHub Actions: Automation
- Git: Version control

## Cloud Platforms

- Microsoft Azure (AKS)
- 

## Conclusion

This architecture provides a robust, scalable, and maintainable MLOps system for heart disease prediction. It follows industry best practices for:

- Reproducibility
- Automation
- Monitoring
- Security
- Scalability

The system is production-ready and can be deployed to various cloud platforms or on-premises infrastructure.