Goal

Design a practical, modular architecture where

IoT devices → Java Spring Boot → Python ML/Visualization → Java Spring Boot → Frontend

with clear protocols, scalable patterns, security, and deployment best practices.

---

1) High-level Architecture (at a glance)

```
[IoT Devices]
   |  (MQTT / HTTPS / WebSocket)
   v
[Ingress Layer]
   - Mosquitto (MQTT Broker) OR Nginx/Ingress (HTTPS/WebSocket)
   - Optional: API Gateway (Spring Cloud Gateway/Traefik)
   |
   v
[Spring Boot Ingest Service]
   - Subscribes to MQTT topics or exposes REST/WebSocket for direct device posts
   - Validates/authenticates, transforms to canonical JSON/Avro
   - Publishes to Kafka/RabbitMQ (async) OR calls Python FastAPI/gRPC (sync)
   |
   +--> (Synchronous path) WebClient/gRPC -> [Python FastAPI ML Service]
   |         - Loads model(s), runs inference/visualization
   |         - Returns results JSON/Protobuf
   |         - Optionally caches features in Redis
   |
   +--> (Asynchronous path) Kafka/RabbitMQ -> [Python Worker(s)]
             - Consumes events, performs batch/stream inference
             - Writes results to Kafka topic / DB (Postgres/Timescale/Influx)
             - Notifies Spring via result topic or callback

[Spring Boot API/UI Service]
   - Aggregates device & inference data (DB + cache)
   - Exposes REST/GraphQL to frontend (React/Thymeleaf)
   - Pushes live updates via Server-Sent Events / WebSocket

[Datastores]
   - Time-series: TimescaleDB/InfluxDB
   - Relational: Postgres/MySQL
   - Cache: Redis
```

```
[Observability]
   - Prometheus + Grafana (metrics)
   - Loki/ELK (logs)
   - OpenTelemetry + Jaeger/Tempo (traces)
```

Recommended default: MQTT for device → server; FastAPI (REST) for initial Spring ↔ Python sync calls; Kafka for scale/out-of-band processing.

---

2) Communication Choices

A. IoT → Spring Boot

- **Preferred: MQTT** via a broker (e.g., **Eclipse Mosquitto**). Devices publish to topics like `devices/{deviceId}/telemetry`.
- **Alternatives:**
- **REST (HTTPS)** when devices are powerful or constrained networks block MQTT.
- **WebSockets** for bi-directional control streams.

Spring Integration options: - Spring for MQTT (via **spring-integration-mqtt** or **Paho** client) to subscribe to topics. - Spring WebFlux for high-throughput REST/WebSocket ingestion.

B. Spring Boot ↔ Python ML

- **Synchronous (low latency, request/response):**
- **REST** (Spring WebClient ↔ **FastAPI**). Simple, JSON.
- **gRPC** (Protobuf schemas) for lower overhead & strict contracts.
- **Asynchronous (throughput, bursty loads):**
- **Kafka** or **RabbitMQ**. Spring publishes events; Python workers consume and write results to result topics/DB. Spring pushes updates to frontend.

Rule of thumb: Start with REST (sync). Add Kafka workers for heavy/expensive models or spikes.

---

3) End-to-End Data Flow

Synchronous (simple, low-latency inference)

1. **Device → Broker**: Device publishes telemetry to `devices/{id}/telemetry` (MQTT over TLS).
2. **Broker → Ingest**: Spring Ingest Service subscribes, validates, transforms to canonical schema.
3. **Ingest → Python**: Spring calls Python FastAPI `/infer` with features JSON.
4. **Python → Ingest**: FastAPI returns `{score, label, anomalies, viz_urls?}`.
5. **Ingest → Store**: Spring persists raw + result (Postgres/Timescale) and caches in Redis.
6. **Ingest → UI**: Spring emits SSE/WebSocket event to UI clients; UI updates charts.

Asynchronous (high-throughput, heavy models)

1–2 same as above. 3. **Ingest → Kafka**: Publish event to `telemetry.raw` (keyed by `deviceId`). 4. **Python Worker**: Consumes, runs model; publishes to `telemetry.inferred` or writes to DB. 5. **Spring API/UI**: Subscribes/consumes result topic or polls DB; pushes to frontend.

---

4) Suggested Project Structure

Java (Spring Boot) — monorepo, multi-module

```
spring-ml-client/
  build.gradle or pom.xml
  settings.gradle
  README.md
  docker/
  k8s/
  common/
    src/main/java/.../common/dto/*
    src/main/java/.../common/config/*
  ingest-service/
    src/main/java/.../ingest/MqttConfig.java
    src/main/java/.../ingest/MqttListener.java
    src/main/java/.../ingest/Validation.java
    src/main/java/.../ingest/Transform.java
    src/main/java/.../ingest/PythonClient.java (WebClient/gRPC)
    src/main/java/.../ingest/Publisher.java (Kafka/RabbitMQ)
    src/main/resources/application.yml
    Dockerfile
  api-ui-service/
    src/main/java/.../api/RestControllers.java
    src/main/java/.../api/Sse/WebSocketHandlers.java
    src/main/java/.../service/QueryService.java
    src/main/java/.../repo/* (JPA/R2DBC)
    src/main/java/.../security/* (JWT/OAuth2)
    src/main/resources/templates/* (Thymeleaf) OR external React app
    Dockerfile
  gateway/
    src/main/java/.../GatewayApplication.java (Spring Cloud Gateway)
    Dockerfile
```

Python (FastAPI + workers)

```
py-ml-backend/
  pyproject.toml or requirements.txt
```

```
app/
  __init__.py
  main.py                # FastAPI app
  api/
    v1/
      schemas.py         # Pydantic models
      routes.py          # /infer, /health, /metrics
  core/
    config.py
    logging.py
  ml/
    models/
      model.pkl or ONNX
    inference.py
    preprocessing.py
  viz/
    plots.py             # optional PNG/HTML generation
  workers/
    consumer.py          # Kafka/RabbitMQ worker
  storage/
    db.py                # Postgres/Timescale/Influx connectors
    cache.py             # Redis
tests/
Dockerfile
k8s/
README.md
```

---

5) Example Technologies & Libraries

- **Spring Boot 3+** (Java 17+), **Spring WebFlux**, **Spring Integration MQTT**, **Spring Kafka**/**AMQP**, **Spring Security** (OAuth2 Resource Server), **MapStruct** (DTO mapping), **Micrometer**.
- **MQTT Broker:** Eclipse **Mosquitto**.
- **Python: FastAPI** (Uvicorn/Gunicorn), **Pydantic**, **NumPy/Pandas/Scikit-learn** or **PyTorch/ONNX Runtime**, **Matplotlib/Plotly** for viz, **confluent-kafka** or **aio-pika** for queues, **SQLAlchemy**.
- **Databases: Postgres/TimescaleDB** (time-series), **InfluxDB** (alt), **Redis** (cache/streams).
- **Message Queue: Kafka** (throughput, partitions), **RabbitMQ** (routing/priority).
- **Frontend:** React (SPA) or Thymeleaf. For live data: SSE or WebSocket.
- **Observability: Prometheus + Grafana**, **ELK/Loki**, **OpenTelemetry**.

---

6) Minimal JSON/Schema Examples

Device → MQTT payload (canonical)

```
{
  "deviceId": "abc-123",
  "ts": "2025-08-26T12:34:56Z",
  "sensors": { "temp": 26.1, "humidity": 58.3, "vibration": 0.12 },
  "meta": { "fw": "1.0.7", "site": "HYD-PLANT-2" }
}
```

Spring → Python /infer (REST)

```
{
  "deviceId": "abc-123",
  "features": { "temp": 26.1, "humidity": 58.3, "vibration": 0.12 },
  "context": { "site": "HYD-PLANT-2" }
}
```

Python → Spring response

```
{
  "deviceId": "abc-123",
  "ts": "2025-08-26T12:34:56Z",
  "score": 0.83,
  "label": "OK",
  "anomalies": [ { "name": "vibration_spike", "severity": "low" } ],
  "explain": { "shap_top": ["vibration", "temp"] }
}
```

Kafka topics (async path)

```
telemetry.raw (key=deviceId, value=canonical JSON/Avro)
telemetry.inferred (key=deviceId, value=result JSON/Avro)
```

---

7) Reference Config/Code Snippets

Spring: MQTT config (application.yml)

```
spring:
  mqtt:
    url: ssl://mosquitto:8883
    clientId: ingest-service
```

```
    username: ${MQTT_USER}
    password: ${MQTT_PASS}
    topics:
      - devices/+/telemetry
```

Spring: WebClient to Python

```
WebClient client = WebClient.builder()
    .baseUrl("http://py-ml:8000")
    .defaultHeader(HttpHeaders.CONTENT_TYPE, MediaType.APPLICATION_JSON_VALUE)
    .build();

Mono<ResultDto> res = client.post()
    .uri("/api/v1/infer")
    .bodyValue(requestDto)
    .retrieve()
    .bodyToMono(ResultDto.class);
```

Python: FastAPI route

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class InferIn(BaseModel):
    deviceId: str
    features: dict
    context: dict | None = None

class InferOut(BaseModel):
    deviceId: str
    ts: str
    score: float
    label: str

@app.post("/api/v1/infer", response_model=InferOut)
async def infer(payload: InferIn):
    score = model.predict_proba(payload.features)
    label = "ALERT" if score < 0.2 else "OK"
    return {"deviceId": payload.deviceId, "ts": datetime.utcnow().isoformat() +
"Z", "score": float(score), "label": label}
```

---

8) Scalability Considerations

- **Backpressure & async:** Use Kafka between ingest and Python for bursts; partition by `deviceId` for order.
- **Autoscaling:** HPA on CPU/RAM/queue lag for Python workers and Spring services.
- **Model loading:** Warm pools of Python workers; use ONNX Runtime or TorchScript for speed; enable batching where applicable.
- **Caching:** Redis for hot features and recent results to reduce model calls.
- **DB design:** Time-series hypertables (Timescale) with retention policies; separate OLTP (Postgres) from analytics (warehouse).
- **Idempotency:** Use event IDs; dedupe on consume.
- **Schema management:** Avro/Protobuf + Schema Registry for Kafka.

---

9) Security Considerations

- **Transport security:** TLS everywhere. MQTT over TLS (8883), REST/gRPC over HTTPS.
- **AuthN/AuthZ:**
- Device credentials (per-device username/password or certs; JWT for HTTPS devices).
- mTLS for device ⇄ broker and service ⇄ service when feasible.
- OAuth2/OIDC (Keycloak/Okta) for user/UI; Spring Resource Server validates JWT.
- **Input validation:** Pydantic (Python), Bean Validation/Jakarta Validation (Spring) with strict schemas & bounds.
- **Topic/route ACLs:** Broker ACLs by topic prefix; least privilege.
- **Secrets mgmt:** Kubernetes Secrets + sealed-secrets/External Secrets; never bake secrets into images.
- **Rate limiting & DoS:** Gateway rate limits; circuit breakers (Resilience4j); retries with jitter.
- **Data hygiene:** PII minimization, encryption at rest (Postgres/TLS, disk encryption), logs scrubbed.

---

10) Deployment & Ops Best Practices

Containers & Orchestration

- **Docker** images per service (Spring Ingest, Spring API/UI, Python API, Python Worker).
- **Kubernetes**:
- Deployments with readiness/liveness probes.
- Services (ClusterIP) + Ingress (TLS) for external traffic.
- **ConfigMaps** for non-secret config; **Secrets** for creds.
- **HPA** for autoscaling; **PodDisruptionBudget** for resilience.
- **Helm** charts or Kustomize for repeatable manifests.

CI/CD

- Build & test on push (Maven/Gradle, pytest).
- Container scan (Trivy/Grype), SAST (CodeQL), SBOM.

- Promote between envs via GitOps (Argo CD/Flux).

Monitoring & Alerting

- **Micrometer** → Prometheus; **FastAPI** metrics via Prometheus middleware.
- **Grafana** dashboards: ingest throughput, model latency, queue lag, error rates, HPA metrics.
- **Logging**: JSON logs; Loki/ELK with structured fields.
- **Tracing**: OpenTelemetry SDKs; propagate trace IDs across Spring ↔ Python calls.

API Versioning & Contracts

- Versioned routes (`/api/v1/...`), OpenAPI/Swagger for Spring & FastAPI.
- Backward-compatible schema evolution; deprecate with timelines.

---

11) Frontend Delivery Pattern

- If using **Thymeleaf**: server-rendered pages consume REST & SSE.
- If using **React**: separate SPA calling `api-ui-service` (BFF pattern). Use WebSocket/SSE for live telemetry and charts (e.g., Recharts/ECharts/Plotly).

---

12) Example Local Dev (Docker Compose)

```
version: "3.9"
services:
  mosquitto:
    image: eclipse-mosquitto:2
    ports: ["1883:1883", "8883:8883"]
    volumes: ["./docker/mosquitto:/mosquitto"]
  kafka:
    image: bitnami/kafka:latest
  postgres:
    image: postgres:16
    environment: ["POSTGRES_PASSWORD=postgres"]
    ports: ["5432:5432"]
  redis:
    image: redis:7
  py-ml:
    build: ./py-ml-backend
    ports: ["8000:8000"]
  ingest-service:
    build: ./spring-ml-client/ingest-service
    ports: ["8081:8081"]
  api-ui-service:
```

```
    build: ./spring-ml-client/api-ui-service
    ports: ["8080:8080"]
```

---

13) Extension Points

- Add **device command & control** topic (`devices/{id}/cmd`) via MQTT; Spring publishes, device subscribes.
- Add **Feature Store** (Feast) if multiple models share features.
- Add **Model registry** (MLflow) and model rollout via tags/AB testing.
- Add **Edge inference** path (on-device model) with periodic cloud validation.

---

14) Quick Start Checklists

MVP (week 1–2): - Mosquitto, Spring Ingest (MQTT) → FastAPI `/infer` → Spring API/UI → SSE to simple dashboard. - Postgres/Timescale for storage, Redis cache.

Scale-up (week 3+): - Introduce Kafka + Python workers. - Add OIDC, HTTPS end-to-end, Prometheus/Grafana, OpenTelemetry. - Package with Helm; deploy to Kubernetes.

TL;DR Defaults to pick now

- Device → Server: **MQTT over TLS** (Mosquitto).
- Spring ↔ Python: **REST (FastAPI)** now; add **Kafka** later.
- DB: **TimescaleDB** + **Redis**.
- Frontend: **React** with SSE for live updates (or Thymeleaf if you prefer server-rendered).
- Observability: **Prometheus + Grafana + Loki**, **OpenTelemetry**.
- Security: **OIDC + JWT**, mTLS where feasible.

This blueprint is battle-tested for real-world IoT + AI systems and is easy to extend as your workloads grow.