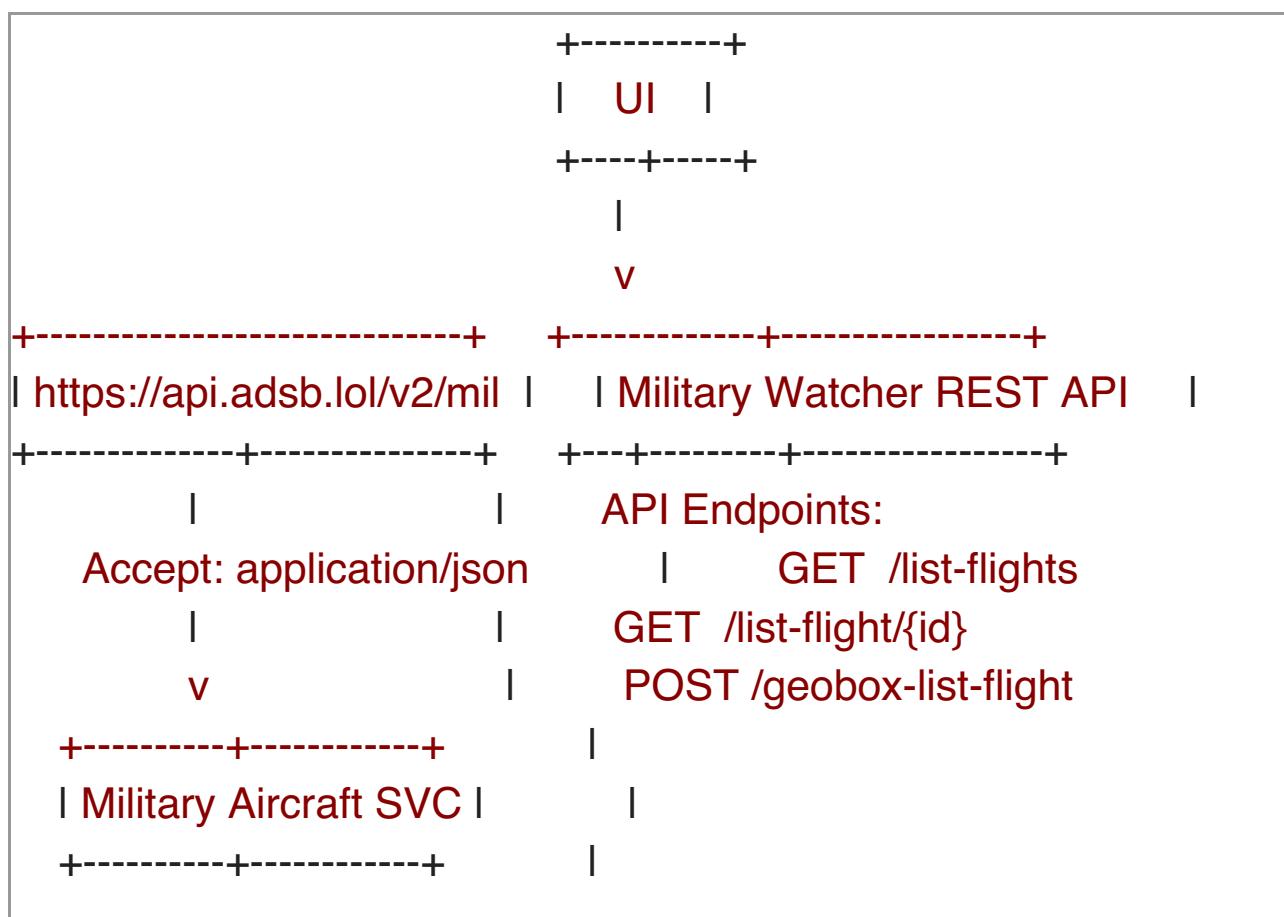


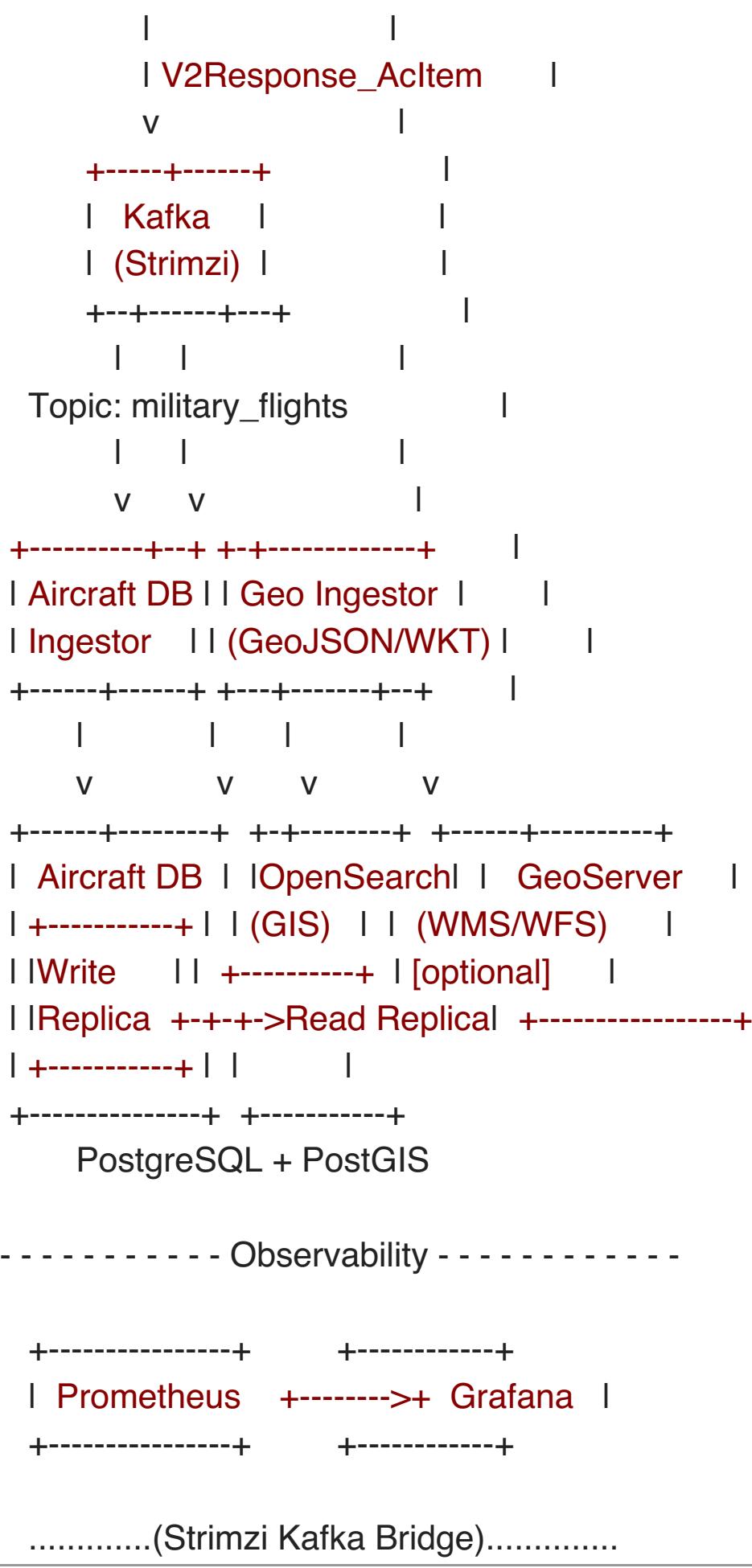
Military Aircraft Tracker - Architecture Document

Overview

The Military Aircraft Tracker is an event-driven system that ingests real-time military aircraft ADS-B data from a public API, streams it through Kafka, persists it to a database with read/write separation, and exposes it via a REST API with a UI frontend. Observability is provided through Prometheus and Grafana.

System Architecture





Components

1. External Data Source - ADS-B Military API

Property	Value
URL	https://api.adsb.lol/v2/mil
Protocol	HTTP/HTTPS
Format	Accept: application/json
Response Model	V2Response_AclItem

The system consumes real-time military aircraft transponder data from the public ADS-B LOL API. This is the sole external data source for the platform.

2. Military Aircraft SVC (Data Ingestion Service)

Role: Polls the ADS-B API and publishes aircraft records to Kafka.

Responsibilities:

- Periodically fetches military aircraft data from <https://api.adsb.lol/v2/mil>
- Deserializes the response into V2Response_AclItem objects
- Publishes records to the military_flights Kafka topic
- Exposes metrics to Prometheus

Metrics Emitted:

Metric	Description
number_of_metrics_retrieved	Total number of aircraft records retrieved

number_of_calls_successful	Count of successful API calls
number_of_calls_unsuccessful	Count of failed API calls
metrics_retrieved_rate	Rate of aircraft records retrieved over time

3. Kafka (Strimzi)

Role: Event streaming backbone.

Property **Value**

Operator Strimzi (Kubernetes-native)

Topic military_flights

Kafka decouples the data ingestion service from the database ingestor, providing:

- Buffering during high-throughput periods
- Replay capability for reprocessing
- Strimzi Kafka Metrics exposed to Prometheus

Strimzi Kafka Bridge is also present in the architecture as an optional HTTP-based access point to Kafka topics.

4. Aircraft DB Ingestor (Consumer Service)

Role: Consumes events from Kafka and writes them to the database.

Responsibilities:

- Subscribes to the military_flights Kafka topic
- Transforms and persists aircraft data to the Aircraft DB (Write

- Replica)
- Exposes metrics to Prometheus

Metrics Emitted:

Metric	Description
number_of_records	Total records written to the database
rate_of_records	Rate of records being ingested over time
number_of_records_ingested	Count of records successfully ingested into the DB
number_of_records_failed_to_ingest	Count of records that failed to ingest into the DB
number_of_records_read_from_kafka	Count of records read/consumed from Kafka

5. Geo Ingestor (GeoJSON/WKT Consumer Service)

Role: Consumes aircraft events from Kafka, transforms them into GeoJSON and WKT geospatial formats, and stores them into a GIS-enabled search/visualization layer.

Responsibilities:

- Subscribes to the military_flights Kafka topic (independent consumer group from the Aircraft DB Ingestor)
- Converts aircraft position data (lat/lon/altitude) into GeoJSON Feature objects with flight metadata as properties
- Converts aircraft position data into WKT POINT / LINESTRING geometry representations
- Indexes GeoJSON documents into OpenSearch with

geo_shape mappings

- Publishes WKT/GeoJSON layers to GeoServer via its REST API (optional)
- Exposes metrics to Prometheus

GeoJSON Output Example:

```
{  
  "type": "Feature",  
  "geometry": {  
    "type": "Point",  
    "coordinates": [-77.0364, 38.8951, 35000]  
  },  
  "properties": {  
    "hex": "AE1234",  
    "flight": "RCH405",  
    "type": "C-17A",  
    "alt_baro": 35000,  
    "gs": 450,  
    "track": 270,  
    "timestamp": "2026-02-27T12:00:00Z"  
  }  
}
```

WKT Output Example:

```
POINT Z(-77.0364 38.8951 35000)  
LINESTRING(-77.0364 38.8951, -77.1200 38.9100, -77.2500  
39.0000)
```

Storage Targets:

Target	Format	Use Case
		Full-text + geospatial search, real-time

OpenSearch GeoJSON dashboards, geo_shape queries

GeoServer WKT

OGC-compliant WMS/WFS map layers,
integration with GIS clients (QGIS, ArcGIS)

Either or both targets can be enabled. OpenSearch is recommended for real-time search and dashboards. GeoServer is recommended when OGC standards compliance or traditional GIS client integration is required.

Metrics Emitted:

Metric	Description
geo_records_read_from_kafka	Count of records consumed from Kafka
geo_records_geojson_created	Count of GeoJSON features successfully created
geo_records_wkt_created	Count of WKT geometries successfully created
geo_records_opensearch_indexed	Count of documents indexed into OpenSearch
geo_records_opensearch_failed	Count of documents that failed to index
geo_records_geoserver_published	Count of features published to GeoServer
geo_records_geoserver_failed	Count of features that failed to publish

6. Aircraft DB (PostgreSQL + PostGIS)

Role: Persistent relational storage with read/write separation and geospatial support.

Component	Purpose
Write Replica	Receives writes from the Aircraft DB Ingestor
Read Replica	Serves read queries from the REST API
Technology Version	Notes
PostgreSQL 17.x	Primary relational database
PostGIS 3.6.x	Geospatial extension for geo queries

The write/read replica pattern enables:

- Write-optimized ingestion without impacting query performance
- Horizontal read scaling for the REST API
- Data replication from write to read replica
- PostGIS geometry columns for spatial queries (ST_Within, ST_DWithin, bounding box)

Suggested Database Schema

```
-- Enable PostGIS
CREATE EXTENSION IF NOT EXISTS postgis;
```

```
-- Aircraft reference data (relatively static)
```

```
CREATE TABLE aircraft (
    id      BIGSERIAL   PRIMARY KEY,
```

```

hex_icao      VARCHAR(6)      NOT NULL UNIQUE, -- 24-bit
ICAO address (e.g. 'AE1234')
registration   VARCHAR(20),          -- tail number (e.g.
'N12345')
aircraft_type  VARCHAR(50),          -- type designator
(e.g. 'C17', 'KC135')
description    VARCHAR(255),          -- full name (e.g.
'Boeing C-17A Globemaster III')
operator       VARCHAR(100),          -- military branch or
unit
country        VARCHAR(100),          -- country of
registration
created_at     TIMESTAMPTZ     NOT NULL DEFAULT NOW(),
updated_at     TIMESTAMPTZ     NOT NULL DEFAULT NOW()
);

```

CREATE INDEX idx_aircraft_hex ON aircraft (hex_icao);

CREATE INDEX idx_aircraft_type ON aircraft (aircraft_type);

-- Flight positions (high-volume, time-series data)

CREATE TABLE flight_positions (

```

id           BIGSERIAL      PRIMARY KEY,
aircraft_id  BIGINT        NOT NULL REFERENCES
aircraft(id),
flight       VARCHAR(8),          -- callsign (e.g.
'RCH405')
position     GEOMETRY(POINTZ, 4326) NOT NULL, --

```

PostGIS 3D point (lon, lat, alt_geom)

alt_baro	INTEGER,	-- barometric altitude (feet)
alt_geom	INTEGER,	-- geometric altitude (feet)
ground_speed	REAL,	-- ground speed (knots)
track	REAL,	-- track angle (degrees from true north)
vertical_rate	INTEGER,	-- climb/descent rate (ft/min)
squawk	VARCHAR(4),	-- transponder squawk code
category	VARCHAR(4),	-- ADS-B emitter category
on_ground	BOOLEAN	DEFAULT FALSE,
seen_at	TIMESTAMPTZ	NOT NULL, -- timestamp from ADS-B source
ingested_at	TIMESTAMPTZ	NOT NULL DEFAULT NOW(),

-- Partitioning key

CONSTRAINT fk_aircraft FOREIGN KEY (aircraft_id)

REFERENCES aircraft(id)

);

-- Spatial index for geographic queries (bounding box, distance, within)

CREATE INDEX idx_flight_positions_geom ON flight_positions

USING GIST (position);

-- Time-based index for recent position queries

CREATE INDEX idx_flight_positions_seen ON flight_positions

(seen_at DESC);

```
-- Composite index for aircraft + time lookups
CREATE INDEX idx_flight_positions_aircraft_time ON
flight_positions (aircraft_id, seen_at DESC);

-- Callsign lookup
CREATE INDEX idx_flight_positions_flight ON flight_positions
(flight);

-- Flight tracks (aggregated linestrings per flight session)
CREATE TABLE flight_tracks (
    id          BIGSERIAL      PRIMARY KEY,
    aircraft_id BIGINT         NOT NULL REFERENCES
aircraft(id),
    flight       VARCHAR(8),           -- callsign
    track_line   GEOMETRY(LINESTRINGZ, 4326),     -- PostGIS
3D linestring of positions
    start_time   TIMESTAMPTZ    NOT NULL,
    end_time     TIMESTAMPTZ,
    point_count  INTEGER        DEFAULT 0,
    created_at   TIMESTAMPTZ    NOT NULL DEFAULT NOW(),
    updated_at   TIMESTAMPTZ    NOT NULL DEFAULT NOW()
);
CREATE INDEX idx_flight_tracks_geom ON flight_tracks USING
GIST (track_line);
CREATE INDEX idx_flight_tracks_aircraft ON flight_tracks
```

```
(aircraft_id, start_time DESC);  
CREATE INDEX idx_flight_tracks_time ON flight_tracks (start_time  
DESC);
```

```
--  
=====  
-- Ingestion metadata (for monitoring / debugging)  
--  
=====  
=====
```

```
CREATE TABLE ingestion_log (  
    id          BIGSERIAL   PRIMARY KEY,  
    batch_id    UUID        NOT NULL,  
    source_url  VARCHAR(255) NOT NULL,  
    records_received INTEGER   NOT NULL DEFAULT 0,  
    records_ingested INTEGER   NOT NULL DEFAULT 0,  
    records_failed  INTEGER   NOT NULL DEFAULT 0,  
    started_at   TIMESTAMPTZ NOT NULL,  
    completed_at TIMESTAMPTZ,  
    error_message TEXT  
);
```

```
CREATE INDEX idx_ingestion_log_time ON ingestion_log  
(started_at DESC);
```

Schema Notes:

- flight_positions.position uses PostGIS GEOMETRY(POINTZ, 4326) — SRID 4326 (WGS84), with Z for altitude
- flight_tracks.track_line aggregates position points into linestrings for trajectory visualization
- GIST indexes enable spatial queries: ST_DWithin(),

ST_Within(), && (bounding box)

- Consider table partitioning on flight_positions by seen_at (e.g., daily/weekly) for time-series performance at scale
 - The ingestion_log table supports the Prometheus metrics for tracking ingestion health
-

7. OpenSearch (Geospatial Search)

Role: Real-time geospatial search and analytics engine.

Technology	Version	Notes
OpenSearch	2.18.x	Distributed search and analytics engine
OpenSearch Geospatial	2.18.x	Plugin for geo_shape and geo_point support
OpenSearch Dashboards	2.18.x	Visualization and map-based dashboards

Capabilities:

- geo_shape field type for indexing GeoJSON geometries
 - geo_point field type for coordinate-based queries
 - Geospatial queries: bounding box, distance, polygon intersection
 - Near real-time indexing for live flight tracking dashboards
 - Full-text search combined with spatial filters
-

8. GeoServer (OGC Map Services – Optional)

Role: OGC-compliant map server for publishing flight data as standard web map layers.

Technology Version	Notes
GeoServer 2.28.x	Open source OGC WMS/WFS/WCS server

Capabilities:

- Publishes flight data as WMS (rendered map tiles) and WFS (vector features)
 - Compatible with GIS desktop clients (QGIS, ArcGIS)
 - Supports GeoJSON and WKT as input/output formats
 - Can connect to PostgreSQL/PostGIS as a data store backend
-

9. Military Watcher REST API

Role: Serves aircraft data to the UI via RESTful endpoints.

Data Source: Reads from the Aircraft DB **Read Replica**.

Endpoints:

Method	Path	Description
GET	/list-flights	List all tracked military flights
GET	/list-flight/{id}	Get details of a specific flight by ID
POST	/geobox-list-flight	List flights within a geographic bounding box

10. UI (Frontend)

Role: User-facing interface for viewing military aircraft data.

Communicates with the Military Watcher REST API to display flight data, likely including map-based visualization given the geographic query support (/geobox-list-flight).

11. Observability Stack

Prometheus

- Scraps metrics from:
 - **Military Aircraft SVC** (API call success/failure rates, retrieval counts)
 - **Kafka / Strimzi** (Strimzi Kafka Metrics)
 - **Aircraft DB Ingestor** (record ingestion counts and rates)
 - **Geo Ingestor** (GeoJSON/WKT creation rates, OpenSearch/GeoServer indexing success/failure)

Grafana

- Visualizes Prometheus metrics via dashboards
- Provides alerting and monitoring for the full pipeline

Data Flow

1. **Military** Aircraft SVC --> **polls** -->
<https://api.adsb.lol/v2/mil>
2. API returns V2Response_AclItem (JSON)
3. **Military** Aircraft SVC --> **publishes** --> Kafka topic:
 military_flights
4. **Aircraft** DB Ingestor --> **consumes** --> Kafka topic:
 military_flights (consumer group A)
5. **Aircraft** DB Ingestor --> **writes** --> Aircraft DB (Write Replica)
6. **Write** Replica --> **replicates** --> Read Replica
7. **Geo** Ingestor --> **consumes** --> Kafka topic:
 military_flights (consumer group B)
8. **Geo** Ingestor --> **converts** --> GeoJSON + WKT
9. **Geo** Ingestor --> **indexes** --> OpenSearch
 (geo_shape documents)
10. **Geo** Ingestor --> **publishes** --> GeoServer
 (WMS/WFS layers) [optional]
11. **Military** Watcher REST API --> **reads** --> Aircraft DB (Read Replica)
12. UI --> **calls** --> Military Watcher REST API

Key Design Decisions

1. **Event-Driven Architecture:** Kafka decouples producers from consumers, enabling independent scaling and fault tolerance.
2. **Read/Write Separation:** Isolates write-heavy ingestion from read-heavy API queries, preventing performance contention.
3. **Strimzi Operator:** Kubernetes-native Kafka management simplifies deployment and operations.
4. **Strimzi Kafka Bridge:** Provides optional HTTP access to Kafka topics without native Kafka clients.

5. **Centralized Observability:** Prometheus + Grafana provide end-to-end pipeline visibility with metrics from every service.
 6. **Dual Geo Pipeline:** The Geo Ingestor runs as an independent Kafka consumer group alongside the Aircraft DB Ingestor, enabling geospatial indexing without impacting relational DB writes.
 7. **OpenSearch for Search + GeoServer for OGC:** OpenSearch provides real-time geospatial search and dashboards; GeoServer provides standards-compliant WMS/WFS for traditional GIS clients.
-

Technology Recommendations & Versions

Java vs Rust — Component Comparison

Criteria	Java (Spring Boot)	Rust (Actix Web / Tokio)
Runtime	High (JIT, GC pauses)	Very High (zero-cost abstractions, no GC)
Performance	possible)	
Memory Footprint	Higher (~200-512 MB per service)	Very Low (~10-50 MB per service)
Kafka Client Maturity	Excellent — native Spring Kafka, Confluent client	Good — rdkafka 0.39 (wraps librdkafka C library)
PostGIS / Geo Support	Excellent — Hibernate Spatial, JTS Topology Suite	Moderate — postgis-diesel or raw SQL via sqlx
Developer	Very Large — extensive libraries, tooling,	Growing — smaller ecosystem, fewer off-the-shelf libs
Ecosystem	community	

Build & Compile Time	Fast (seconds)	Slow (minutes for full builds)
Deployment Size	Larger (~150-300 MB container images)	Very Small (~20-50 MB static binaries)
TLS Support	Built-in (JSSE, Spring Security)	Built-in (rustls or openssl crate)
Learning Curve	Moderate — widely known	Steep — ownership model, lifetimes
Observability	Excellent — Micrometer (Prometheus) + Spring Actuator	Good — prometheus crate, manual integration
Concurrency Model	Virtual Threads (Java 25), reactive (WebFlux)	Async/await with Tokio runtime

Recommendation

Component Recommended	Rationale
Military Aircraft SVC	Spring Kafka provides mature, battle-tested Kafka producer integration. Micrometer gives turnkey Prometheus metrics.
Aircraft DB Ingestor	Hibernate Spatial / Spring Data JPA provide first-class PostGIS support for writing geospatial data. Spring Kafka consumer is well-proven. GeoTools library provides robust GeoJSON/WKT generation.
Geo Ingestor	OpenSearch Java client and GeoServer REST client are mature. Spring Kafka consumer reuse across ingestors.

Military Watcher REST API

Either

Java offers Hibernate Spatial for geo queries out of the box. Rust offers lower latency and smaller footprint if the team has Rust experience.

Summary: Java is the pragmatic default for all four services due to superior Kafka, PostGIS, and geospatial ecosystem support (GeoTools, OpenSearch client). Rust is viable for the REST API if the team prioritizes raw performance and has Rust expertise, but adds friction for geospatial queries.

Java Framework Comparison — Spring Boot vs Quarkus vs Micronaut vs Vert.x vs Helidon vs Plain Java

Framework Overview

Framework	Version	Maintainer	Philosophy
Spring Boot	4.0.x	VMware/Broadcom	Full-featured, convention-over-configuration, largest ecosystem
Quarkus	3.30.x	Red Hat	Kubernetes-native, GraalVM-first, supersonic startup
Micronaut	4.x	Oracle (OCI)	Compile-time DI, reflection-free, minimal overhead
Eclipse	5.0.x	Eclipse Foundation	Reactive event-loop toolkit,

Vert.x			polyglot, low-level control
Helidon	4.x	Oracle	Virtual threads-first (Nima server), MicroProfile support
Plain Java	JDK 25	Oracle / OpenJDK	No framework — direct use of JDK APIs + standalone libraries

What "Plain Java" Means

A frameworkless approach using only the JDK and standalone libraries:

Concern	Plain Java Library
HTTP Server	jdk.httpserver (built-in) or Jetty / Undertow embedded
REST Routing	Manual path matching or Javalin (thin wrapper over Jetty)
Dependency Injection	None — constructor wiring, factory pattern, or manual service locator
Configuration	java.util.Properties, env vars, or Typesafe Config
JSON	Jackson ObjectMapper (direct usage)
Kafka	Apache Kafka Client (org.apache.kafka:kafka-clients)
Database	JDBC + HikariCP connection pool
ORM	None — raw SQL via JDBC, or standalone Hibernate (no container)
HTTP Client	java.net.http.HttpClient (built-in since JDK 11)
Scheduling	ScheduledExecutorService or java.util.Timer
Logging	SLF4J + Logback
Build	Gradle or Maven (same as framework projects)

Performance Characteristics

Metric	Spring Boot 4	Quarkus 3.30	Micronaut 4	Vert.x 5	Helidon 4	Play Java
JVM Startup	~1.9s	~1.1s	~0.7s	~0.5s	~0.8s	~0.1-0.3s
Native Startup	~0.10s	~0.05s	~0.05s	N/A (no AOT)	~0.06s	~0.03s
JVM Heap (idle)	~150-200 MB	~80-120 MB	~60-100 MB	~30-60 MB	~50-80 MB	~15-20 MB
Native Heap (idle)	~50-80 MB	~12-30 MB	~15-35 MB	N/A	~20-40 MB	~5-10 MB
Throughput (req/s)	High	Very High	Very High	Highest	High	Very High
GraalVM Native	Supported	First-class	First-class	Limited	Supported	(no reflection issue)

Kafka Integration

Framework	Kafka Library	Style	Maturity
Spring Boot	Spring Kafka (wraps Apache client)	Annotation-driven @KafkaListener	Excellent
Quarkus	SmallRye Reactive Messaging	Reactive Streams, @Incoming/@Outgoing	Excellent
Micronaut	Micronaut Kafka	Annotation-driven @KafkaListener	Good
Vert.x	vertx-kafka-client	Reactive/callback,	Good

		manual		
Helidon	MP Reactive Messaging or direct client	MicroProfile or manual	Moderate	
Plain Java	kafka-clients (Apache) directly	KafkaProducer / KafkaConsumer poll loop	Excellent (it's the underlying library)	

Plain Java Kafka Detail:

All framework Kafka integrations wrap the same org.apache.kafka:kafka-clients library. Using it directly means:

- Full control over consumer poll loop, offset commits, rebalance listeners
- No annotation magic — explicit Properties config, manual thread management
- TLS/SASL configured via standard Kafka Properties (security.protocol, ssl.*, sasl.*)
- Trade-off: you must build your own consumer lifecycle, error handling, and graceful shutdown

PostgreSQL + PostGIS Support

Framework	DB Access	PostGIS / Geo Support
Spring Boot	Spring Data JPA + Hibernate	Hibernate Spatial — first-class geo types
Quarkus	Hibernate ORM / Panache + Reactive PG	Hibernate Spatial via Quarkus Hibernate ORM
Micronaut	Micronaut Data + Hibernate / JDBC	Hibernate Spatial (via Hibernate integration)

Vert.x	Reactive PG Client (non-ORM)	Manual — raw SQL with PostGIS functions
Helidon	JPA (EclipseLink) or JDBC	Limited — manual PostGIS via raw SQL
Plain Java	JDBC + HikariCP (or standalone Hibernate)	Raw SQL: ST_SetSRID(ST_MakePoint(?, ?, ?), 4326), or standalone Hibernate Spatial

Plain Java DB Detail:

- **JDBC approach:** Use PreparedStatement with PostGIS WKT/WKB directly — e.g.,
ST_GeomFromText('POINT(-77.03 38.89)', 4326)
- **Standalone Hibernate:** Hibernate + Hibernate Spatial can run without Spring/Quarkus — requires manual SessionFactory bootstrap via persistence.xml or programmatic config
- **Connection pooling:** HikariCP is the standard standalone pool (also used internally by Spring/Quarkus)
- Trade-off: no repository pattern, no declarative transactions — all SQL and connection management is explicit

Observability (Prometheus Metrics)

Framework	Metrics Library	Setup Effort
Spring Boot	Micrometer + Actuator	Zero-config
Quarkus	SmallRye Metrics / Micrometer	Minimal (extension)
Micronaut	Micrometer Micrometer or	Minimal (module)
Vert.x	manual	Manual wiring

Helidon	MicroProfile Metrics	Minimal
	Prometheus Java Client	Manual — register counters/gauges, expose /metrics
Plain Java	(simpleclient)	HTTP endpoint

Plain Java Metrics Detail:

```
// io.prometheus:simpleclient + simpleclient_httpserver
Counter recordsIngested = Counter.build()
    .name("number_of_records_ingested")
    .help("Records successfully ingested").register();

// Expose on port 9090
HTTPServer metricsServer = new HTTPServer(9090);
```

- Requires manually creating and registering each metric
- Must serve a /metrics endpoint (via simpleclient_httpserver or your own HTTP handler)
- No auto-instrumentation for JVM metrics unless you add simpleclient_hotspot

TLS Support

Framework	TLS Configuration
Spring Boot	application.yml SSL properties, Spring Security
Quarkus	application.properties TLS config, built-in
Micronaut	application.yml SSL config, built-in
Vert.x	Programmatic NetServerOptions / HttpServerOptions
Helidon	Config-file or programmatic TLS setup SSLContext + KeyStore via JSSE APIs, or

Plain Java Jetty/Undertow SSL config

Plain Java TLS Detail:

```
// Standard JSSE for Kafka / HTTP clients
SSLContext sslContext = SSLContext.getInstance("TLS");
KeyManagerFactory kmf =
KeyManagerFactory.getInstance("SunX509");
kmf.init(keyStore, password);
sslContext.init(kmf.getKeyManagers(), trustManagers, null);
```

- Full control but verbose — 15-20 lines of boilerplate vs 3 lines of YAML in frameworks
- For Kafka: set ssl.keystore.location, ssl.truststore.location in Properties
- For HTTP server: configure SSLContext on Jetty ServerConnector or Undertow Builder

Developer Experience

Criteria	Spring Boot	Quarkus	Micronaut	Vert.x
Learning Curve	Low (widely known)	Low-Moderate	Low-Moderate	Moderate-High
Documentation	Extensive	Excellent	Good	Good
Community Size	Very Large	Large	Medium	Medium

Extension Ecosystem	Massive (starters)	Large (400+ extensions)	Growing	Moderate (modules)
Live Reload / Dev Mode	Spring DevTools	Quarkus Dev Mode (best-in-class)	Incremental compile	Manual
IDE Support	Excellent	Very Good	Good	Good
Boilerplate	Low	Low	Low	Medium
Testing	Spring Test + Testcontainers	Quarkus Test + DevServices	Micronaut Test	Manual setup
Time to Production	Fast	Fast	Fast	Moderate

Per-Component Recommendation

	Component	Spring Boot	Quarkus	Micronaut	Vert.x	Helidon	Plain Java
Military Aircraft SVC	Good	Best	Good	Viable	Viable	Viable	Viable
Aircraft DB Ingestor	Best	Good	Good	Viable	Limited	Viable	
Geo Ingestor	Best	Good	Viable	Viable	Limited	Limited	

Military Watcher REST API	Good	Best	Good	Viable	Good	Limited
----------------------------------	------	-------------	------	--------	------	---------

Rationale:

- **Spring Boot** is the strongest choice for the **DB Ingestor** and **Geo Ingestor** due to Hibernate Spatial and the GeoTools/OpenSearch Java client integrations working seamlessly with Spring's DI and transaction management.
- **Quarkus** is the strongest choice for the **Military Aircraft SVC** and **REST API** — its SmallRye Reactive Messaging provides elegant Kafka integration, and its fast startup + low memory make it ideal for Kubernetes. It also has Hibernate Spatial support via its Hibernate ORM extension, making it viable for all components.
- **Micronaut** is a solid alternative across the board with comparable performance to Quarkus, but has a slightly smaller ecosystem for geospatial libraries.
- **Vert.x** offers the highest raw throughput but requires the most manual wiring — best suited for teams experienced with reactive/event-loop programming.
- **Helidon** is a clean virtual-threads option but has the smallest community and limited geospatial support.
- **Plain Java** is detailed below.

Plain Java — Detailed Assessment

Where Plain Java works well:

- **Military Aircraft SVC** — This service is conceptually simple: poll an HTTP API on a schedule, deserialize JSON, produce to Kafka. The JDK's built-in HttpClient + Jackson +

KafkaProducer + ScheduledExecutorService is straightforward. Minimal wiring needed. Virtual threads (JDK 25) handle concurrency without a framework.

- **Aircraft DB Ingestor** — A Kafka consumer loop that writes to PostgreSQL. KafkaConsumer poll loop + JDBC PreparedStatement with PostGIS SQL is manageable. HikariCP handles pooling. This is viable but you lose declarative transaction management and must handle offset commits, retries, and error handling manually.

Where Plain Java struggles:

- **Geo Ingestor** — Requires integrating GeoTools (GeoJSON/WKT), OpenSearch Java Client, and optionally GeoServer REST Client alongside Kafka. Without DI, coordinating these library lifecycles, configuration, and error handling becomes significant boilerplate. Rated **Limited**.
- **Military Watcher REST API** — Building a production REST API without a framework means implementing routing, content negotiation, request validation, error handling, CORS, pagination, and health checks from scratch. Using Javalin or bare Jetty reduces this but still requires manual plumbing for database access, connection pooling, and request/response serialization. Rated **Limited**.

Plain Java — Pros & Cons Summary:

Pros	Cons
Fastest startup and lowest memory of any Java option	Significant boilerplate for DI, lifecycle, config
No framework lock-in or	No declarative transactions, no auto-

upgrade risk	configuration
Easiest GraalVM native compilation (no reflection)	Must implement health checks, graceful shutdown, etc.
Full control, no "magic"	Testing requires manual setup (no test slices)
Fewer dependencies = smaller attack surface	No dev mode / live reload built-in
Simple to understand and debug	Every cross-cutting concern (logging, metrics, TLS, error handling) wired by hand
Good for single-purpose, long-running daemons	REST API development is tedious without routing framework

Verdict: Plain Java is a reasonable choice for **simple, single-purpose services** like the Military Aircraft SVC (HTTP poll + Kafka produce) where the overhead of a framework exceeds the complexity of the service itself. For services with multiple integrations (Geo Ingestor) or REST API surface area (Watcher REST API), a framework saves substantial development and maintenance time. If choosing plain Java, consider using **Javalin** (~3 MB) as a minimal HTTP routing layer — it adds REST routing without framework overhead.

If not using Spring Boot: Quarkus is the recommended alternative. It provides comparable Kafka and PostGIS support through SmallRye and Hibernate extensions, with significantly better startup time, lower memory usage, and first-class Kubernetes/GraalVM native support. It also has an excellent developer experience with its live-coding dev mode.

Recommended Technology Versions

Backend Services (Java Stack – Spring Boot)

Technology	Version	Notes
Java	JDK 25 (LTS)	Latest LTS, virtual threads, premier support through 2030
Spring Boot	4.0.x	Built on Spring Framework 7, Java 25 first-class support
Spring Kafka	(included in Boot 4)	Kafka producer/consumer with auto-configuration
Micrometer	(included in Boot 4)	Prometheus metrics via Spring Actuator
Hibernate Spatial	7.x	PostGIS geometry type support for JPA entities
GeoTools	32.x	GeoJSON/WKT generation, geometry operations
OpenSearch Client	2.18.x	Java client for indexing GeoJSON into OpenSearch
Build Tool	Gradle 8.x or Maven	

Backend Services (Java Stack — Quarkus Alternative)

Technology	Version	Notes
Java	JDK 25 (LTS)	Latest LTS, virtual threads
Quarkus	3.30.x	Kubernetes-native, GraalVM native image support
SmallRye Reactive Messaging	(included)	Kafka producer/consumer via @Incoming / @Outgoing
Quarkus Hibernate ORM	(included)	Hibernate Spatial support for PostGIS geometry types
SmallRye Metrics / Micrometer	(included)	Prometheus metrics endpoint
GeoTools	32.x	GeoJSON/WKT generation, geometry operations
OpenSearch Client	2.18.x	Java client for indexing GeoJSON into OpenSearch
Build Tool	Gradle 8.x or Maven	Quarkus Maven/Gradle plugin for dev mode and native builds

Backend Services (Plain Java — No Framework)

Technology	Version	Notes
Java	JDK 25 (LTS)	Virtual threads for concurrency, built-in HttpClient
Javalin (optional)	6.x	Minimal HTTP routing (~3 MB), wraps Jetty
Jetty (alt)	12.x	Embedded HTTP server if not using Javalin
Jackson	2.18.x	JSON serialization/deserialization

kafka-clients	3.9.x	Apache Kafka producer/consumer (direct usage)
HikariCP	6.x	JDBC connection pool
PostgreSQL JDBC	42.x	JDBC driver
Hibernate (optional)	7.x	Standalone ORM + Hibernate Spatial (no Spring/Quarkus container)
GeoTools	32.x	GeoJSON/WKT generation, geometry operations
OpenSearch Client	2.18.x	Java client for indexing GeoJSON into OpenSearch
Prometheus simpleclient	0.16.x	Counters, gauges, histograms + HTTP exposition server
SLF4J + Logback	2.x / 1.5.x	Logging
Typesafe Config (optional)	1.4.x	HOCON/properties-based configuration
Build Tool	Gradle 8.x or Maven	

Backend Services (Rust Stack – if chosen)

Technology	Version	Notes
Rust	1.85+	Latest stable toolchain
Actix Web	4.12	Highest performance Rust web framework
Tokio	1.x	Async runtime
rdkafka	0.39	Async Kafka client wrapping librdkafka
SQLx	0.8	Async, compile-time checked SQL (PostGIS via raw queries)

Diesel (alt)	2.x	ORM with postgis-diesel crate for typed geo support
rustls	0.23	Pure-Rust TLS (or openssl crate for OpenSSL bindings)
prometheus	0.13	Prometheus metrics exposition

Message Broker

Technology	Version	Notes
Strimzi Operator	0.50.x	Latest stable, Kubernetes-native Kafka operator
Apache Kafka	3.9+ / 4.0	Managed by Strimzi

Strimzi Kafka Security Options:

Security Mode	Description	Use Case
TLS	Mutual TLS (mTLS) — certificate-based authentication and encryption SASL/SCRAM-SHA-512 —	Production, zero-trust environments
SASL	username/password over encrypted channel	Multi-tenant, simpler credential mgmt
PLAIN	SASL/PLAIN — plaintext username/password (use only with TLS transport)	Development, internal networks

Recommendation: Use **TLS** (mTLS) for production inter-service communication. SASL/SCRAM over TLS is a good alternative when certificate management overhead is a concern. Avoid PLAIN without TLS encryption in any environment.

Database

Technology	Version	Notes
PostgreSQL	17.x	Latest stable major release, robust and proven
PostGIS	3.6.x	Geospatial extension for geographic queries

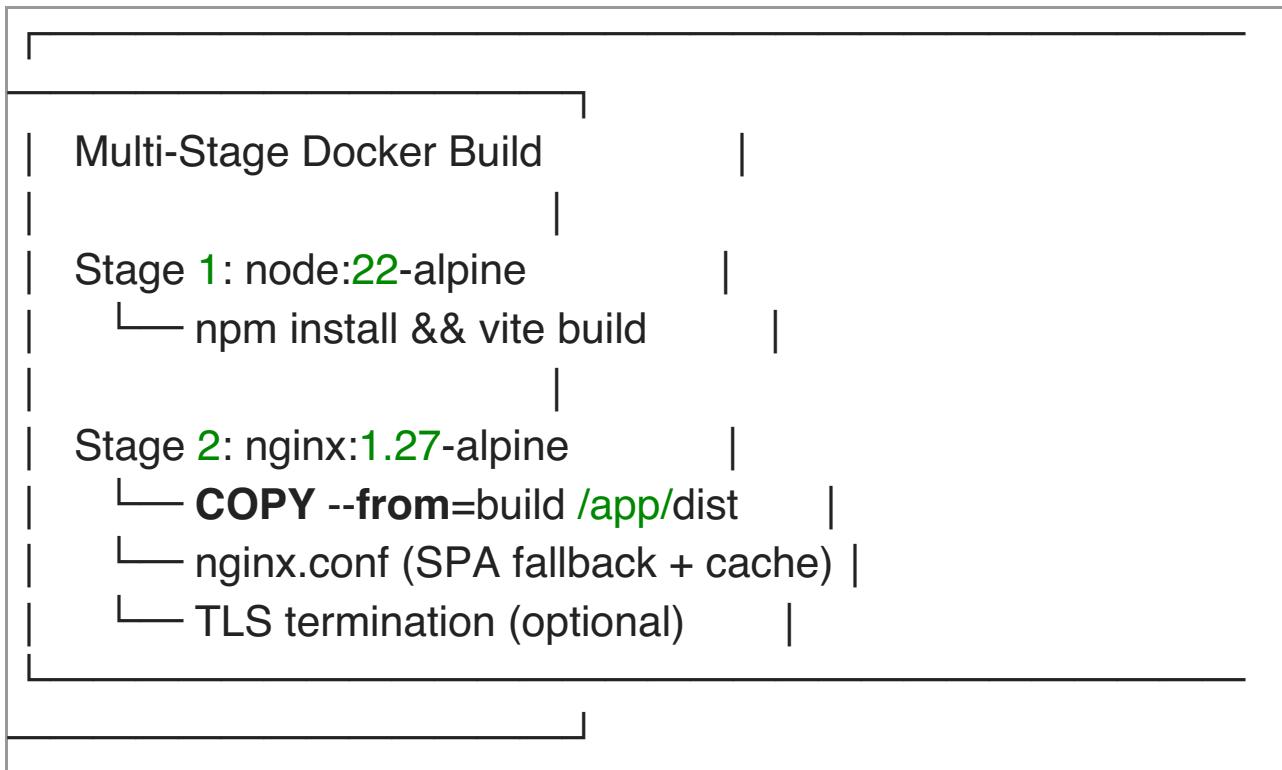
Replication Setup:

- **Write Replica:** Primary PostgreSQL instance receiving inserts from the Ingestor
- **Read Replica:** Streaming replication replica serving the REST API
- PostGIS must be installed on both replicas

Frontend

Technology	Version	Notes
React	19.2.x	Latest stable with Server Components, concurrent features
Vite	6.x	Build tooling — fast HMR in dev, optimized production builds
TypeScript	5.x	Type safety for frontend code
Nginx	1.27.x	HTTP container serving the built React SPA

UI Container Architecture:



Service-Level TLS

Service	TLS Option
Military Aircraft SVC	TLS for outbound HTTPS to api.adsb.lol, mTLS to Kafka
Aircraft DB	mTLS to Kafka, TLS to PostgreSQL
Ingestor	(sslmode=verify-full)
Geo Ingestor	mTLS to Kafka, TLS to OpenSearch, TLS to GeoServer REST API
Military Watcher REST API	TLS termination (direct or via ingress), TLS to PostgreSQL
UI (Nginx)	TLS termination at Nginx or Kubernetes Ingress
Strimzi Kafka	Inter-broker TLS, client mTLS or SASL/SCRAM over TLS
PostgreSQL	ssl = on, client certificate auth optional

OpenSearch	TLS for transport and REST layers, optional client cert auth
GeoServer	TLS termination (reverse proxy or direct)

Technology Summary

Layer	Technology / Service	Version
Data Source	ADS-B LOL API (api.adsb.lol)	v2
Ingestion Service	Military Aircraft SVC (Spring Boot)	JDK 25 / Boot 4.0
Message Broker	Apache Kafka (Strimzi on Kubernetes)	Strimzi 0.50
DB Consumer	Aircraft DB Ingestor (Spring Boot)	JDK 25 / Boot 4.0
Geo Consumer	Geo Ingestor (Spring Boot + GeoTools)	JDK 25 / Boot 4.0
Database	PostgreSQL + PostGIS (Write + Read Replicas)	PG 17 / PostGIS 3.6
Geo Search	OpenSearch + Geospatial Plugin	OpenSearch 2.18
Map Server	GeoServer (optional, OGC WMS/WFS)	GeoServer 2.28
API Layer	Military Watcher REST API (Spring Boot)	JDK 25 / Boot 4.0
Frontend	React + Vite + TypeScript (Nginx container)	React 19.2 / Vite 6
Monitoring	Prometheus + Grafana	
Kafka Access	Strimzi Kafka Bridge (optional)	