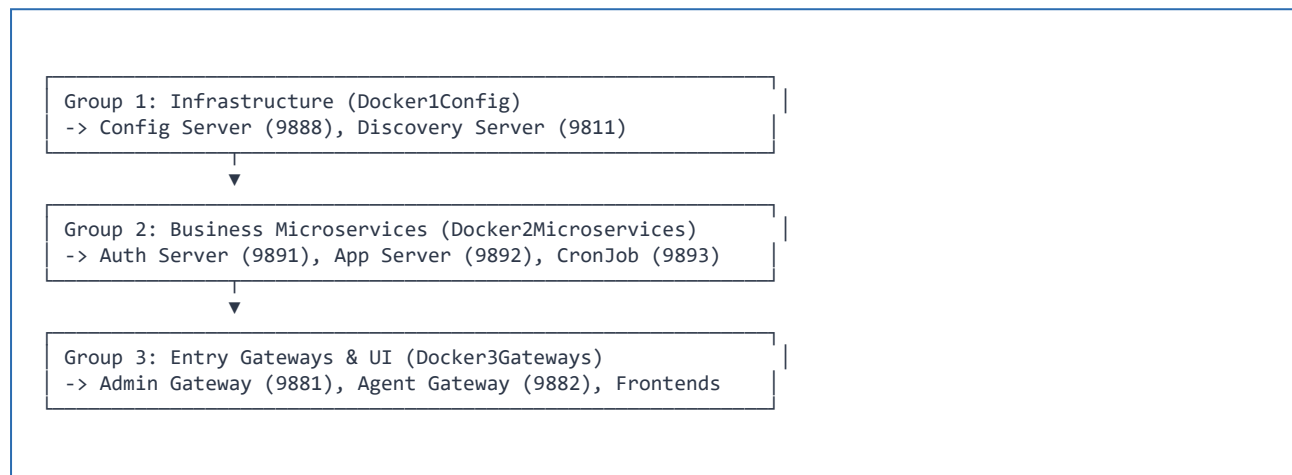


9. DEPLOYMENT ARCHITECTURE

The Immigration Management System utilizes **Docker Containerization** with **Docker Compose** to achieve environment parity and modular scalability. This architecture abstracts the host operating system, ensuring consistent behavior from developer machines to production servers.

9.1 Docker Architecture & Compose Groups

The deployment is structured into three logical groups based on initialization dependencies. Services must be started in this specific sequence to ensure the microservice mesh stabilizes correctly.



9.2 Deployment Workflow

The standard workflow follows a **Build-Transfer-Start** pattern:

1. **Artifact Generation:** Compile the project using `mvn clean install` to generate SNAPSHOT JARs.
2. **Staging:** Transfer JAR files to their respective `DockerDir` directories on the host server.
3. **Infra Boot:** Run `docker-compose up -d` in `Docker1Config`. Verify Eureka dashboard is accessible at port 9811.
4. **Service Boot:** Deploy `Docker2Microservices`. This group maps host volumes for `/root/applogs` and the `geo.json` config.
5. **Edge Boot:** Deploy `Docker3Gateways` once core services are registered in Eureka.

9.3 Runtime Dependencies

- **External:** Oracle Database 18c (Standard JDBC connectivity), SMTP Relay for email, and wkhtmltopdf binary.
- **Internal:** Every service requires the Config Server to be active to resolve properties via `spring.config.import`.

9.4 Port Mapping & Environment Configuration

The following table outlines the networking configuration for the microservice cluster.

SERVICE NAME	INTERNAL	EXTERNAL	CONFIGURATION VARIABLES
Config Server	9888	9888	<code>spring.cloud.config.server.git.uri</code>
Discovery	9811	9811	<code>eureka.client.service-url.defaultZone</code>
Authorization	9891	9891	<code>spring.datasource.url</code> (Auth Schema)
Application	9892	9892	<code>WKHTML_RUN</code> , <code>app.init.geo-json-path</code>
CronJob	9893	9893	<code>spring.profiles.active</code>
Admin Gateway	9881	9881	<code>CORS allowed-origins</code>

9.5 Environment Separation

- **Development (dev):** Local Oracle instance, h2 for testing, DEBUG logging level enabled for rapid debugging.
- **Alpha (alpha):** Staging environment. Uses specific Git branch configurations. Log level set to INFO.
- **Production (prod):** Hardened environment with Oracle 18c Enterprise. Log level set to WARN. HTTPS is enforced at the Gateway level.

9.6 Container & Logging Configuration

Resource Allocation

To ensure stability, Docker resource limits are recommended:

- **App Server:** 2.0 CPUs / 2.0GB Memory.
- **Auth Server:** 1.0 CPU / 1.0GB Memory.

Logging Strategy

Containers are ephemeral, so logs are mounted to host volumes (`./applogs:/root/applogs`). Logs are formatted using SLF4J and Logback, rotating daily to prevent disk overflow.

10. DEVOPS AND CI/CD

While current deployment is managed through sequential scripts, the architecture is primed for a fully automated CI/CD lifecycle.

10.1 Version Control & Artifact Generation

- **Bitbucket Integration:** Code is managed in a multi-module Git repository. Configuration properties are stored in a separate `immigration-properties.git` repository for security separation.
- **Maven Lifecycle:** Uses `ImmigrationParent` as the master POM to maintain version consistency across all dependencies (Spring Boot, Resilience4j, etc.).
- **Docker Images:** Built using `openjdk:8-jdk-alpine` to minimize image size and attack surface.

10.2 Recommended CI/CD Pipeline

To transition to modern DevOps, the following 7-stage pipeline is recommended:

STAGE	ACTION
1. Source	Trigger on Git push to develop or main.
2. Build	<code>mvn clean compile</code> all modules.
3. Test	Execute JUnit 5 suite and JaCoCo coverage reports.
4. Package	Generate JARs and build Docker images tagged by version.
5. Artifact	Push images to a private Docker Registry (e.g., Nexus, ECR).
6. Staging	Deploy to Alpha environment for QA verification.
7. Prod	Blue-Green deployment to production after manual approval.

10.3 Configuration Management (Git-Ops)

The **Config Server** follows a Git-Ops philosophy. Changes to database passwords, feature flags, or API keys are committed to the property repository and refreshed across the microservice mesh via the `/actuator/refresh` endpoint without requiring a full service restart.

11. LOGGING AND OBSERVABILITY

In a distributed environment, observing the health of individual nodes is not enough; the system tracks the journey of traveler requests across the entire service mesh.

11.1 Application Logging & Error Tracking

- **Standard:** Logback implementation with RollingFileAppender.
- **Structure:** [timestamp] [level] [traceId] [spanId] [message].
- **Exceptions:** Global error handling via `@ControllerAdvice` ensures business exceptions (e.g., `CustomResponseException`) return meaningful HTTP status codes and JSON payloads instead of stack traces.

11.2 Distributed Tracing with Sleuth & Zipkin

The system integrates **Spring Cloud Sleuth** to inject Trace IDs into every request header. These IDs propagate through Feign clients to downstream services.

- **Zipkin (2.2.2.RELEASE):** Visualizes the latency of each service call. If a ticket generation fails, Zipkin identifies exactly which hop (Gateway → App → PDF Generator) caused the failure.

11.3 Metrics & Troubleshooting

Spring Boot Actuator

Active monitoring of JVM memory, GC, and HTTP throughput via `/actuator/metrics`. Health checks at `/actuator/health` are used by Docker Compose for service health probes.

Troubleshooting Matrix

SCENARIO	DIAGNOSTIC ENDPOINT	COMMON SOLUTION
Token Invalid	<code>/oauth/check_token</code>	Check Auth Schema connectivity; verify BCrypt salt.
PDF Failed	<code>/actuator/health</code>	Check WKHTML_RUN binary path and permissions.
Service Invisible	Eureka Dashboard (9811)	Verify Config Server is providing the Eureka URL.