◆ SDLC ARCHITECTURE BLUEPRINT V1.0

VibeCode AI

A Lovable.dev clone — full-stack AI SaaS platform built to master Spring Boot, Microservices, Kafka, Docker, Kubernetes & Spring AI.

PROJECT TYPE AI SaaS Platform	ARCHITECTURE Microservices
SERVICES 8 Core + 3 Infra	LEARNING SCOPE Full Cohort 4.0

---

01 SDLC Flow

🔍 PLANNING	🏗️ DESIGN
💻 DEVELOPMENT	🧪 TESTING

🚀 DEPLOYMENT	📊 MONITORING

We start with **Planning & Design** — you're here right now. This document defines the full system before a single line of code is written. This mirrors how real engineering teams work and is itself resume-worthy.

---

02 System Architecture Layers

FRONTEND

`React 18` `TypeScript` `Vite` `TailwindCSS` `Monaco Editor` `WebSockets`

API GATEWAY

`Spring Cloud Gateway` `JWT Validation` `Rate Limiting` `SSL Termination`

MICROSERVICES

`Auth Service` `User Service` `Project Service` `AI Generation Service`
`Code Executor Service` `Billing Service` `Notification Service`

AI / LLM LAYER

Spring AI | OpenAI GPT-4o | RAG Pipeline | MCP Server | pgvector (Embeddings) | Prompt Templates

MESSAGING

Apache Kafka | Kafka Topics | Redis Pub/Sub | Schema Registry

DATA LAYER

PostgreSQL | MySQL | MongoDB | Redis Cache | pgvector | Spring Data JPA

INFRASTRUCTURE

Docker | Kubernetes (EKS) | Eureka Service Registry | Spring Cloud Config | ELK Stack | Zipkin | AWS CodePipeline

---

03 Microservice Descriptions

PORT: 8081 | eureka: auth-service

Auth Service

Handles all user identity: registration, login, JWT issuance, Google OAuth2, token refresh, logout, password encoding (BCrypt). Acts as the single source of truth for authentication across all services.

Spring Security 6 | JWT | Google OAuth2 | MySQL | Spring Data JPA

Key Endpoints

POST /auth/register POST /auth/login POST /auth/refresh

PORT: 8082 | eureka: user-service

User Service

Manages user profiles, subscription tiers (Free / Pro / Enterprise), usage quotas, role/permission management. Stores usage history for billing calculations.

Spring Boot | Spring Data JPA | PostgreSQL | Swagger/OpenAPI

Key Endpoints

GET /users/{id} PUT /users/{id}/plan GET /users/{id}/usage

PORT: 8083 | eureka: project-service

Project Service

CRUD for AI-generated projects. Stores project metadata, prompt history, file tree, versions, and public/private visibility. Persists generated source code as structured records. Supports forking.

Spring Boot | MongoDB | Spring Data MongoDB | Actuator

Key Endpoints

POST /projects GET /projects/{id}/files POST /projects/{id}/fork

PORT: 8084 | eureka: ai-generation-service

AI Generation Service

The brain of VibeCode. Receives natural language prompts, enriches them via RAG (retrieves relevant code patterns from the vector DB), invokes OpenAI via Spring AI, and streams generated code back. Implements MCP for tool-use (file creation, linting, running commands).

Spring AI | OpenAI GPT-4o | RAG | MCP Server | pgvector | WebFlux

Key Endpoints

POST /generate/stream POST /generate/chat POST /generate/embed

PORT: 8085 | eureka: code-executor-service

Code Executor Service

Sandboxed execution environment. Receives generated code, spins up isolated Docker containers, runs the code, captures output/errors, and returns results. Critical for preview functionality. Uses Java Executor Framework for async scheduling.

Docker SDK | Java Executor Framework | Schedulers | WebSockets | Spring Boot

Key Endpoints

POST /execute | GET /execute/{id}/status | WS /execute/{id}/logs

PORT: 8086 | eureka: billing-service

Billing Service

Tracks token usage per user, enforces plan limits, handles Stripe webhook events for subscription lifecycle, generates invoices, manages credit top-ups. Listens to Kafka events from the AI Generation Service to count usage.

Spring Boot | Kafka Consumer | PostgreSQL | Stripe API | REST Template

Key Endpoints

GET /billing/{userId}/usage | POST /billing/webhooks/stripe

PORT: 8087 | eureka: notification-service

Notification Service

Listens to Kafka topics for events (generation complete, quota exceeded, payment success) and sends real-time notifications via WebSocket push, email (SendGrid), and in-app. Implements Redis Pub/Sub for live UI updates.

Kafka Consumer | Redis Pub/Sub | WebSockets | SendGrid API | Spring Boot

Key Topics Consumed

`generation.complete` `user.quota.exceeded` `payment.success`

PORT: 8080 | entry point for all traffic

API Gateway

Single entry point. Validates JWT tokens, applies rate limiting (Resilience4J), routes to downstream services via Eureka discovery, handles CORS and SSL termination. Auto-refreshes config via Spring Cloud Bus.

`Spring Cloud Gateway` `Eureka Client` `Resilience4J` `Spring Cloud Bus` `Redis Rate Limiting`

Pattern

`Client → Gateway → Eureka Discovery → Service`

---

04 End-to-End Request Flow

This is the path a user request takes from "type a prompt" to "see generated code" — every step maps to a topic in your Spring Boot cohort.

01

User logs in via React Frontend

React app calls `POST /api/auth/login` through the API Gateway. Gateway validates and passes to **Auth Service**. BCrypt password check → JWT issued → stored in HttpOnly cookie. Google OAuth flow also handled here. *Topics covered: Spring Security, JWT, Google OAuth, Session Management.*

02

Frontend opens WebSocket connection

After login, the React client opens a persistent WebSocket to receive real-time events (generation progress, notifications). **Notification Service** subscribes to Redis Pub/Sub and pushes to this socket. *Topics: Redis Pub/Sub, WebSockets.*

03

User types a prompt & hits Generate

`POST /api/generate/stream` → API Gateway validates JWT, applies rate limiting → routes to **AI Generation Service** via Eureka. The Gateway uses Spring Cloud Config for dynamic routing rules. *Topics: API Gateway, Eureka, Rate Limiting, Resilience4J.*

04

RAG Pipeline enriches the prompt

AI Generation Service converts the prompt to an embedding (via OpenAI Embeddings API through Spring AI). Searches **pgvector** for similar code patterns from a pre-indexed knowledge base. Retrieves top-K chunks, injects them into the prompt as context. *Topics: Spring AI, RAG, Embeddings, Vector DB.*

05

LLM call via Spring AI + MCP

Enriched prompt + context sent to OpenAI GPT-4o via Spring AI's chat client. MCP Server exposes tools: `create_file`, `run_command`, `install_dependency`. The model orchestrates these tools to generate a full project. Prompt templates ensure structure. *Topics: Spring AI, MCP, Prompt Engineering, LLM Orchestration.*

06

Generated code streamed to frontend

Spring WebFlux streams the AI response in real-time (SSE / chunked response) so the Monaco editor displays code token-by-token, just like Lovable. *Topics: Spring WebFlux, Reactive APIs, Streaming.*

07

Kafka event published: generation.complete

Once complete, AI Generation Service publishes a `generation.complete` event to Kafka with payload {userId, projectId, tokensUsed, timestamp}. *Topics: Kafka Publisher, Kafka Topics, Schema Registry.*

08

Billing Service consumes the event

Billing Service (Kafka Consumer) picks up the event, deducts tokens from user quota, checks limits, writes to PostgreSQL. If quota exceeded, publishes `user.quota.exceeded` event. *Topics: Kafka Consumer, Spring Data JPA, Database Transactions.*

09

Code stored & preview executed

Generated code persisted to MongoDB via **Project Service**. User clicks "Run" → **Code Executor Service** spins up an isolated Docker container, runs the code, streams logs over WebSocket. *Topics: MongoDB, Docker SDK, Java Executor Framework, Cron Jobs.*

10

Notification pushed & logs aggregated

Notification Service sends in-app notification. All services push structured logs to **ELK Stack** (Elasticsearch + Logstash + Kibana). Zipkin traces the full distributed request. Spring Boot Actuator exposes health metrics. *Topics: ELK, Zipkin, Actuator, Centralized Logging.*

05 Technology Map

CONCERN	TECHNOLOGY	WHERE USED
AI / LLM	Spring AI, OpenAI GPT-4o, MCP	AI Generation Service
RAG	pgvector, Spring AI Embeddings	AI Generation Service
Authentication	Spring Security 6, JWT, Google OAuth2	Auth Service, API Gateway
API Gateway	Spring Cloud Gateway, Resilience4J	API Gateway (port 8080)
Service Discovery	Netflix Eureka (Spring Cloud)	All microservices
Messaging	Apache Kafka, Schema Registry	AI Gen → Billing → Notification
Caching	Redis (Spring Cache + Pub/Sub)	Gateway, Notification, Project
Primary Databases	PostgreSQL (Auth, Billing, Users)	Auth, User, Billing Services
Document Store	MongoDB	Project Service (file trees, code)
Vector Database	pgvector extension on PostgreSQL	AI Generation Service

CONCERN	TECHNOLOGY	WHERE USED
Real-time	WebSockets, Redis Pub/Sub, SSE	Frontend ↔ Notification / Code Exec
ORM	Spring Data JPA, Hibernate	Auth, User, Billing Services
Containerization	Docker, Docker Compose	All services (local dev + executor)
Orchestration	Kubernetes (AWS EKS)	Production deployment
CI/CD	AWS CodePipeline, CodeDeploy	Build → Test → Deploy pipeline
Distributed Tracing	Spring Cloud Sleuth + Zipkin	All microservices
Centralized Logging	ELK Stack (Elasticsearch+Logstash+Kibana)	All microservices
Config Management	Spring Cloud Config Server + Bus	All microservices (external config)
Health & Metrics	Spring Boot Actuator, Prometheus	All microservices
Documentation	Swagger / OpenAPI 3	All microservices
AOP	Spring AOP (Before/After/Around)	Logging, security auditing, metrics
Testing	JUnit 5, Mockito, Testcontainers	All services (unit + integration)
Frontend	React 18, TypeScript, Monaco Editor	Web client

06

Resume & Interview Value

This project is deliberately designed to cover every topic in the cohort in a production context. Here's what you can claim on your resume:

Headline Project Bullet

"Built VibeCode AI — a Lovable.dev clone, a microservices-based AI SaaS platform using Spring Boot, Spring AI, Apache Kafka, Docker, and Kubernetes. Implemented RAG pipeline with pgvector for context-aware code generation, JWT-secured APIs, CI/CD on AWS, and distributed tracing with Zipkin."

Spring Boot 3 | Spring AI + RAG | MCP Server | Microservices | Apache Kafka

Docker | Kubernetes | Spring Security 6 | JWT + OAuth2 | Spring Cloud

PostgreSQL + pgvector | MongoDB | Redis Cache | ELK Stack | Zipkin

AWS CI/CD | JUnit + Mockito | Spring AOP | System Design | React + TypeScript

---

07

Recommended Build Order

Follow this order to learn progressively — each service builds on the previous one's concepts:

W1

Week 1 — Foundation

Set up monorepo. Create Spring Boot projects for each service skeleton. Configure Maven, Spring Initializr, Eureka registry. Deploy Hello World to Docker Compose.

W2

Week 2-3 — Auth + User Service

Build Auth Service: registration, login, JWT, Google OAuth, BCrypt. Then User Service: profiles, JPA, PostgreSQL. Connect both through API Gateway.

W3

Week 4 — Project Service + MongoDB

Project CRUD with MongoDB. File tree structure. Learn NoSQL vs SQL tradeoffs in context.

W4

Week 5-6 — AI Generation Service

Integrate Spring AI + OpenAI. Build RAG pipeline with pgvector. Implement MCP server. Stream responses to frontend.

W5

Week 7 — Kafka + Billing + Notifications

Wire up Kafka event flow: generation → billing → notification. Redis Pub/Sub for real-time push.

W6

Week 8 — Code Executor Service

Docker-in-Docker sandboxing, Java Executor Framework, WebSocket log streaming.

W7

Week 9-10 — Testing, AOP, Observability

JUnit + Mockito unit tests. AOP for cross-cutting concerns. ELK + Zipkin + Actuator setup.

W8

Week 11-12 — Kubernetes + CI/CD

Kubernetes deployment manifests. HPA for autoscaling. AWS CodePipeline for full CI/CD.
