CodeVibe Spec2App – Natural Language to Modular App Generator

Design Document (v1.0) – Senior Software Architecture

1) Overview

CodeVibe Spec2App transforma especificaciones en lenguaje natural (ej.: “crear app para reportar baches en Miami”) en artefactos de ingeniería y un código base ejecutable. El sistema está compuesto por agentes interoperables: Analyst (NLP → requisitos + dominios) y Coder (generación de arquitectura, UML, API y scaffolding en React + Node). En versiones posteriores se añaden Modeler (UML/ERD experto), DocWriter (API/README/ADR), y QA (tests + validación).

2) Goals / Non-Goals

Goals: • Conversión robusta de requerimientos en blueprint arquitectónico modular. • Generación automática de artefactos: Arquitectura, UML textual, OpenAPI, WBS, ADRs. • Scaffold de monorepo con React (Vite) + Node (NestJS/Express), TS, ESLint, Prettier, Jest. • MVP demostrable con 2 agentes (Analyst y Coder) antes de diciembre. Non■Goals (v1): • Soporte multi-plataforma móvil nativo (iOS/Android) más allá de PWA. • Optimización ML a bajo nivel; se usarán LLMs/Semantic tools externas. • Generación de infra IaC completa (solo boilerplate base de Docker/Compose).

3) Personas & Use Cases

Personas: Product Owner, Arquitecto, Full■stack Dev, Consultor GovTech. Use cases clave: UC■1: PO ingresa un enunciado y obtiene arquitectura + APIs + código base listo para correr. UC■2: Arquitecto sube un enunciado extendido (epics + reglas de negocio) y obtiene UML/ERD + ADRs. UC■3: Dev genera un módulo adicional (ej.: “módulo de autenticación SSO + OIDC”).

4) System Architecture (MVP)

Arquitectura por capas y agentes: • Ingestion API (REST) recibe el prompt de especificación. • Analyst Agent: extracción de dominios, entidades, casos de uso, riesgos, NFRs, contexto. • Orchestrator: compone tasks, valida consistencia, produce un “Design Contract” (JSON). • Coder Agent: genera artefactos (OpenAPI, UML textual, carpetas, código TS React/Node). • Artifact Store: Git monorepo + almacenamiento de diseños (JSON, YAML, Markdown). • Preview Runner: docker-compose para levantar backend + frontend y validar endpoints. • Docs Portal: documentación (OpenAPI UI, Storybook opcional, Markdown).

4.1) Interoperabilidad entre agentes (Design Contract)

Interfaz canónica JSON que describe el sistema destino. Es la “fuente de verdad” intercambiada entre agentes. Campos mínimos: - metadata: name, domain, locale, constraints. - nonFunctionals: security, scalability, availability, perf, compliance. - entities: name, attributes (type, required, validation), relations. - services: name, operations (input/output), errors, auth, rateLimit. - apis: OpenAPI skeleton por servicio. - ui: pages, routes, components, state. - ci: pipelines esperados, quality gates. - risks: lista con mitigaciones. - wbs: entregables y tareas.

Ejemplo (fragmento Design Contract):

```
{ "metadata": {"name":"PotholeReporter","locale":"en-US"},
"entities":[{"name":"Report","attrs":[{"name":"id","type":"uuid"},{"name":"location","type":"geo"},
{"name":"photoUrl","type":"string"},{"name":"status","type":"enum:OPEN|IN_PROGRESS|CLOSED"}]}], "ser
vices":[{"name":"ReportService","operations":[{"name":"createReport","in":"ReportInput","out":"Repor
```

```
t"}]}], "ui":{"routes":["/","/reports/:id"],"components":["ReportForm","ReportList","MapView"]} }
```

5) Componentes Detallados

5.1 Ingestion API (NestJS/Express): • POST /spec → valida payload, crea job, retorna jobId. • GET /spec/:id/status → progreso (ingestion, analysis, codegen, docs, done). • Seguridad: API Key + optional OAuth2 client_credentials. Rate limit y audit log. 5.2 Analyst Agent: • Tareas: NER/NLU para entidades, reglas de negocio, NFRs; genera backlog (epics→stories). • Salida: Design Contract v1 + matriz de trazabilidad (Requisito→Entidad/Servicio/API). 5.3 Orchestrator: • Verifica consistencia (nombres, tipos, duplicados); normaliza vocabulario y taxonomía. • Aplica plantillas de arquitectura (hexagonal, clean architecture) según dominio. 5.4 Coder Agent: • Genera OpenAPI (YAML), UML textual (PlantUML/Mermaid), código TS (React+Node). • Produce tests iniciales (Jest + Supertest) y seeds de datos. 5.5 Artifact Store: • Monorepo (pnpm workspaces) + convención de carpetas; versionado semántico; cambios vía MR. 5.6 Preview Runner: • Dockerfile multi-stage; docker-compose.dev.yml con hot reload; script make run-dev. 5.7 Docs Portal: • Swagger UI, Redocly; README, ADRs, WBS; opcional Storybook para UI.

6) Tech Stack & Standards

• Lenguaje: TypeScript end■to■end. • Backend: NestJS (preferente) o Express con Zod + tRPC opcional. • Frontend: React + Vite + React Query + Zustand/Redux Toolkit; UI library (Shadcn/Tailwind). • DB: PostgreSQL (Prisma ORM). GIS opcional: PostGIS. • Infra: Docker, docker-compose, Makefile; opcional Terraform v2. • CI/CD: GitLab CI/GitHub Actions; pnpm; caches; quality gates (lint, test, typecheck). • QA: Jest, Vitest, Playwright (smoke e2e), Supertest. • Observabilidad: Pino + OpenTelemetry + Prometheus/Grafana (futuro). • Seguridad: OWASP ASVS, JWT/OIDC, RBAC/ABAC, input validation con Zod/class-validator. • Docs: OpenAPI 3.1, ADRs, Markdown; PlantUML/Mermaid textual.

7) Estructura de Monorepo

```
/ ■■ apps/ ■ ■■ api/ (NestJS) ■ ■■ web/ (React+Vite) ■■ packages/ ■ ■■ contracts/ (Design Contract schemas, zod) ■ ■■ ui-kit/ (componentes compartidos) ■ ■■ sdk/ (client TS autogenerado desde OpenAPI) ■■ tools/ (codegen, scripts, generators) ■■ docs/ (OpenAPI, ADRs, WBS, UML) ■■ .github/ or .gitlab/ ■■ package.json / pnpm-workspace.yaml
```

8) Estrategia de Generación de Código

Pipeline: 1) Analyst → Design Contract (JSON). 2) Orchestrator → normaliza + valida → Contract v1.1. 3) Coder: • OpenAPI YAML (por servicio). • Server stubs (NestJS controllers/services). • Prisma schema + migrations. • React routes/pages/components + servicios API. • Tests: unit + e2e smoke. 4) Docs: README, ADR-0001 (Arquitectura), WBS.md, UML.mmd/uml. 5) Compose: docker-compose.dev.yml + env.example. Reglas: • Idempotencia: regenerar sin romper ediciones del usuario (áreas “protected” con tags). • Plantillas Mustache/EJS + AST transforms (TS■Morph) para precisión. • Convenciones: nombres kebab-cased para rutas, PascalCase para tipos, id: uuid v7.

9) UML (textual – para posterior render)

Mermaid – Diagrama de clases (ejemplo):

```
classDiagram class Report { +uuid id +Geo location +string photoUrl +ReportStatus status +datetime createdAt } class User { +uuid id +string email +string role } ReportStatus : OPEN ReportStatus : IN_PROGRESS ReportStatus : CLOSED User "1" --> "many" Report : creates
```

Mermaid – Secuencia (ejemplo flujo MVP):

```
sequenceDiagram participant Client participant IngestionAPI participant Analyst participant Orchestrator participant Coder participant Repo Client->>IngestionAPI: POST /spec IngestionAPI->>Analyst: analyze(spec) Analyst-->>Orchestrator: designContract v1 Orchestrator-->>Coder: contract v1.1 Coder-->>Repo: commit scaffolding + docs IngestionAPI-->>Client: status + repo URL
```

10) API Contracts (REST OpenAPI 3.1 – resumen)

```
POST /spec body: { spec: string, options?: { locale, stackPreset, constraints[] } } resp: { jobId, status } GET /spec/{id}/status resp: { status: enum[queued|analysis|codegen|docs|done|error], progress:0..100 } GET /artifact/{id} resp: zip/links to repo artifacts Auth: API-Key (x-api-key) + rate limit (IP/Key).
```

11) Seguridad y Compliance

• Validación inputs (Zod) + límites de longitud y estructura (anti■prompt injection). • Sandboxing para codegen (contenedor aislado, seccomp, no network salvo mirrors permitidos). • Secrets fuera del repo; .env.example con placeholders; soporte a SOPS/1Password. • Auditoría y trazabilidad: jobId, actor, timestamp, hashes de artefactos. • Dependabot/Renovate + SBOM (syft) + SCA (trivy). • Política de datos: no almacenar especificaciones sensibles; retención configurable.

12) Calidad, CI/CD y Observabilidad

• Pipelines: lint → typecheck → unit → e2e smoke → build. • Coverage gates (>=80% en core generators). • Pre-commit (lint-staged + prettier). • Versionado semántico + Conventional Commits + CHANGELOG automático. • Logs estructurados (Pino) + trazas OTel en orquestador (futuro). • Métricas: tiempo total de generación, fallas de consistencia, diffs protegidos tocados.

13) WBS (MVP → diciembre)

Fase 0: Fundaciones (1-2 semanas) - Monorepo, tooling, CI básico, plantillas, contrato JSON, validadores. Fase 1: Analyst (2 semanas) - NER/NLU prompts + extracción de entidades, NFRs, backlog. Fase 2: Orchestrator (1 semana) - Normalización, validaciones, selección de patrón arquitectónico. Fase 3: Coder (3 semanas) - Generación OpenAPI, NestJS/Prisma, React/Vite, tests smoke. Fase 4: Docs & Demo (1 semana) - README, ADR, WBS, UML textual, docker-compose demo. Entregable demo: especificación “baches en Miami” → repo ejecutable en <10 min.

14) Riesgos y Mitigaciones

R1: Ambigüedad en lenguaje natural → Formular “Design Contract” con preguntas aclaratorias automáticas. R2: Inconsistencia entre módulos generados → Validadores AST + pruebas smoke. R3: Sobre-escritura de código del usuario → Zonas protegidas + etiquetas // … R4: Costos LLM → cache y embeddings locales para repetición; modos “lite”. R5: Seguridad en codegen → sandbox + linters de seguridad + SCA en CI.

15) Mejores Prácticas

• Clean Architecture/Hexagonal, SOLID, DRY, 12■Factor App, Defensive Programming. • Tipado estricto TS, errores typed (Result/Either), guards centrales. • Eventos de dominio (mediator/pub■sub interno) para desacoplar. • Feature flags para estrategias de generación. • Documentar decisiones en ADRs versionadas. • Contratos primero (Contract■First): OpenAPI → SDK cliente → implementación.

16) Apéndice – Prompt Templates (resumen)

Analyst Prompt (resumen): - Input: especificación NL, contexto, restricciones. - Output: Design Contract v1 + backlog (epics→stories), NFRs, riesgos. Coder Prompt (resumen): - Input: Design Contract v1.1 - Output: OpenAPI, server stubs, Prisma, React scaffold, tests, docs.

— Fin del documento —