



## **Maestría en Inteligencia Artificial Aplicada**

### **Proyecto Integrador**

#### **Avance 3: Generador de Aplicaciones de Base de Datos Mediante Tecnologías de Inteligencia Artificial**

Profesores:

Dra. Grettel Barceló Alonso

Dr. Luis Eduardo Falcón

Dr. Guillermo Mota Medina

Equipo 59

A01795457	Renzo Antonio Zagni Vestrini
-----------	------------------------------

A01362405	Roger Alexei Urrutia Parke
-----------	----------------------------

A01795501	Héctor Raúl Solorio Meneses
-----------	-----------------------------

## Table de Contenido

<b>Resumen de los avances de la semana.....</b>	<b>2</b>
<b>Arquitectura del sistema.....</b>	<b>3</b>
Componentes y responsabilidades.....	3
Estado compartido (BuildState).....	4
<b>Metodología.....</b>	<b>4</b>
Planificación con LLM.....	4
Generación determinista vs. asistida.....	4
Documentacion.....	4
<b>Implementación.....</b>	<b>4</b>
Tecnologías.....	4
Puntos de extensión.....	4
Manejo de errores.....	5
Caso de uso.....	5
<b>Apéndice A — Instrucciones de Ejecución.....</b>	<b>5</b>
<b>Apéndice B — Documentación del Sistema (Readme.md).....</b>	<b>6</b>
Sistema Agentic para la generación programática de aplicaciones fullstack a partir de lenguaje natural.....	6
Descripción general.....	7
Arquitectura del sistema.....	7
Tecnologías principales.....	8
Instalación y uso rápido.....	8
🔧 Pre-requisitos.....	8
Instalación.....	8
Ejecución del Builder.....	9
Base de datos y backend.....	9
Estructura del proyecto.....	9
API Reference (ejemplo).....	10
POST /api/items.....	10
Estado compartido (BuildState).....	11
Extensiones futuras.....	11
Evaluación propuesta.....	11
Pruebas.....	12

## Resumen de los avances de la semana

Durante la presente semana se logró el desarrollo de los servicios requeridos para establecer la conexión con el sistema de OpenAI, así como la definición de los agentes responsables de la generación del código correspondiente a las distintas capas de la aplicación: front-end (Vue.js), back-end (Express y Node.js) y gestión de base de datos (PostgreSQL). Asimismo, se avanzó en la formulación y ajuste de los *prompts* necesarios para la generación automatizada del código fuente de la aplicación final.

El desarrollo de este proyecto ha evidenciado un elevado nivel de complejidad técnica y un alto grado de detalle en su implementación. Para el Avance 4, se ha establecido como objetivo principal optimizar los *prompts* del *back-end* y del *front-end*, incorporar la funcionalidad CRUD (crear, leer, actualizar y eliminar registros), y dar los primeros pasos hacia la generación de aplicaciones de mayor complejidad. Estas nuevas aplicaciones buscarán trascender la estructura de formularios simples, permitiendo la gestión integral de sistemas con relaciones de tipo maestro–detalle, características propias de entornos empresariales o administrativos avanzados.

## Arquitectura del sistema

El flujo principal se implementa como un grafo de estados:

plan → scaffold → db → backend → frontend → review → finalize

- plan (Planner): transforma el prompt en un plan JSON (app\_name, entities, fields).
- scaffold: crea la estructura del proyecto y dependencias base.
- db: genera esquemas PostgreSQL por entidad (plantilla o LLM).
- backend: produce rutas CRUD Express/Node con acceso pg.
- frontend: crea componentes Vue (SFC) para formularios y listas.
- review: compone README.md con guía de inicio.
- finalize: imprime resumen y conteo de artefactos.

## Componentes y responsabilidades

- Gestor de prompts (PromptSpec): admite system, user y examples y permite carga dinámica desde ./prompts/\*.yaml con fallbacks sensatos.

- Parser robusto JSON: `_parse_json_safely` tolera fences (````json`), texto extra y selecciona el primer bloque `{...}` válido, elevando errores claros.
- Persistencia de artefactos: `write_file` garantiza creación de rutas y escritura idempotente.
- Estructura de proyecto: `backend/`, `frontend/`, `ops/`, `models/`, `routes/`.
- Backend base: `server.js` con auto-mount de routers, `db.js` con pool pg, `package.json`, `.env.sample`.
- DevOps: `ops/docker-compose.yml` para levantar Postgres localmente.

## Estado compartido (BuildState)

Estructura tipada que consolida: prompt de usuario, plan, directorio base, artefactos por categoría, logs y errores.

## Metodología

### Planificación con LLM

El nodo Planner usa un LLM (configurable) y un prompt tipo `system designer` que exige salida estricta en JSON con tipos SQL. Se incluye un ejemplo `few-shot` sobre inventario y proveedores.

### Generación determinista vs. asistida

Para DB, Backend y Frontend existen dos rutas:

- Determinista (plantillas): garantiza reproducibilidad y tiempos constantes.
- Asistida por LLM: activa si existen prompts YAML específicos (`db`, `backend`, `frontend`), facilitando estilos o convenciones distintas.

### Documentación

El agente Reviewer agrega un `README.md` con stack, instrucciones de arranque y ubicación de rutas API, uniformando la experiencia de onboarding.

## Implementación

### Tecnologías

- Orquestación: LangGraph (máquina de estados, nodos y aristas explícitas).
- Razonamiento: LangChain + ChatOpenAI (modelos configurables por variable de entorno).
- Backend: Node.js + Express, pg (PostgreSQL).
- Frontend: Vue SFC con data binding y fetch a `/api/<entity>`.

- DevOps: Docker Compose para Postgres.
- Configuración: .env y .env.sample para secrets y puertos.

## Puntos de extensión

- Frameworks: Vue↔React, Express↔FastAPI/Django, SQL↔NoSQL.
- Esquemas: Integración de migraciones (Knex/Prisma/Alembic), claves foráneas, índices, constraints.
- Frontend: validación de formularios, paginación, filtros, routing, estado global.
- Seguridad: autenticación (OAuth/JWT), autorización (RBAC/ABAC), rate limiting, CORS estricto.
- Observabilidad: structured logging, métricas, tracing del grafo.

## Manejo de errores

- Parser JSON tolerante: evita fallas por formatting drift en respuestas del LLM.
  - Fallbacks deterministas: si no hay prompts YAML, se usa generación por plantilla.
  - Logs: el estado final imprime resumen de artefactos y bitácora de nodos.
- 

## Caso de uso

Prompt: “Build an application to manage inventory items and suppliers. All screens should support CRUD.”

Salida esperada:

- backend/models/Item.sql y Supplier.sql con campos tipados (TEXT/INTEGER).
- backend/routes/Item.js y Supplier.js con endpoints CRUD.
- frontend/src/components/ItemManager.vue / SupplierManager.vue con formulario + lista.
- ops/docker-compose.yml para Postgres y README.md con guía de arranque.

## Apéndice A — Instrucciones de Ejecución

### 1) Instalar dependencias (lado Python)

```
pip install langgraph langchain-openai python-dotenv pyyaml
```

### 2) Configurar credenciales LLM

```
export OPENAI_API_KEY=...
```

### 3) Ejecutar el builder

```
python agentic_builder.py "Build me an application to manage  
inventory items"
```

### 4) Provisionar Postgres

```
cd ops && docker compose up -d
```

### 5) Iniciar backend

```
cd ../backend  
cp .env.sample .env  
npm install  
node server.js
```

### 5) Iniciar frontend

```
cd ../frontend  
npm run dev
```

## Apéndice B — Documentación del Sistema (Readme.md)

*Sistema Agentic para la generación programática de aplicaciones fullstack a partir de lenguaje natural*

### Descripción general

Agentic App Builder es un sistema modular que convierte instrucciones en lenguaje natural en una aplicación fullstack completamente funcional.

A través de un pipeline orquestado con LangGraph y potenciado por LLMs, el sistema:

1. Planifica entidades y relaciones de datos.
2. Genera esquemas PostgreSQL.
3. Construye una API REST con Express/Node.js.
4. Crea componentes Vue tipo CRUD.
5. Produce documentación inicial y estructura operativa lista para ejecución.

El enfoque busca demostrar cómo los *agentes orquestados* pueden combinar planificación, generación y auditoría de manera reproducible y verificable en contextos de innovación y enseñanza.

### Arquitectura del sistema

graph LR

A[plan] --> B[scaffold]

B --> C[db]

C --> D[backend]

D --> E[frontend]

E --> F[review]

F --> G[finalize]

Cada nodo representa una etapa de generación:

Nodo	Rol principal
plan	Transforma el prompt en un plan JSON con entidades, campos y tipos.
scaffold	Crea la estructura base del proyecto y dependencias.
db	Genera esquemas PostgreSQL por entidad.
backend	Implementa rutas CRUD con Express y acceso a <a href="#">pg</a> .
frontend	Crea componentes Vue (formularios y listas).
review	Redacta <a href="#">README.md</a> inicial con guía de uso.
finalize	Genera resumen, logs y conteo de artefactos.

## Tecnologías principales

Capa	Tecnología
Orquestación	<a href="#">LangGraph</a>
Razonamiento	LangChain + ChatOpenAI
Backend	Node.js + Express + pg
Frontend	Vue (Single File Components)
Base de datos	PostgreSQL
DevOps	Docker Compose
Configuración	<a href="#">.env</a> / <a href="#">.env.sample</a>

## Instalación y uso rápido

### Pre-requisitos

- Python 3.10+
- Node.js 18+



- Docker 24+
- Clave de API de OpenAI ([OPENAI\\_API\\_KEY](#))

## Instalación

### # 1. Clonar el repositorio

```
git clone https://github.com/tuusuario/agentic-app-builder.git  
cd agentic-app-builder
```

### # 2. Instalar dependencias Python

```
pip install langgraph langchain-openai python-dotenv pyyaml
```

### # 3. Configurar credenciales LLM

```
export OPENAI_API_KEY="tu_clave_aqui"
```

## Ejecución del Builder

```
python agentic_builder.py "Build an app to manage inventory and suppliers"
```

## Base de datos y backend

```
cd ops
```

```
docker compose up -d
```

```
cd ../backend
```

```
cp .env.sample .env
```

```
npm install
```

```
node server.js
```

La aplicación quedará disponible en <http://localhost:3000>.

## Estructura del proyecto

```
agentic-app-builder/
```

```
|— backend/
| |— models/      # Esquemas SQL generados
| |— routes/      # Endpoints CRUD Express
| |— db.js        # Conexión con Postgres
| |— server.js    # Servidor base
| |— package.json
|— frontend/
| |— src/components # Componentes Vue tipo CRUD
|— ops/
| |— docker-compose.yml
|— prompts/      # Plantillas YAML opcionales
|— agentic_builder.py
|— README.md
```

## API Reference (ejemplo)

POST [/api/items](#)

Request

```
{
  "name": "Laptop",
  "quantity": 10
}
```

Response

```
{
  "id": 1,
  "name": "Laptop",
  "quantity": 10
}
```

}

## Estado compartido (BuildState)

El sistema mantiene un contexto tipado que conserva:

- Prompt original del usuario.
- Plan JSON estructurado.
- Artefactos generados.
- Logs y errores detectados.
- Resultados finales (resumen y conteo).

## Extensiones futuras

Área	Mejora propuesta
Seguridad	Autenticación OAuth/JWT, RBAC/ABAC, rate limiting.
Observabilidad	Logging estructurado, métricas y tracing del grafo.
DB	Migraciones (Knex/Prisma/Alembic), claves foráneas e índices.
Frontend	Validación, paginación, filtros y estado global (Pinia/Vuex).
Frameworks alternos	Vue ↔ React, Express ↔ FastAPI/Django.

## Evaluación propuesta

Métrica	Descripción
Eficiencia	Tiempo de scaffolding vs. baseline manual.
Calidad de artefactos	Tasa de compilación y análisis estático.
Fidelidad	Cobertura de entidades vs. estándar anotado.
Robustez del parser	Tolerancia a ruido en respuestas LLM.

Experiencia UX (dev)      Tiempo de onboarding al proyecto.

Se propone una evaluación ciega entre versiones deterministas y asistidas por LLM, con análisis estadístico (t-test o Mann–Whitney U).

## Pruebas

# Pruebas unitarias Python

pytest tests/

# Linter y compilación Node

npm run lint

npm run test

Resultados esperados:

- 100% lint pass rate.
- < 60 segundos de scaffolding promedio.
- Parser JSON con  $\geq 95\%$  éxito en ruido controlado.

## Roadmap

- Implementar autenticación JWT.
- Agregar validación de formularios Vue.
- Integrar métricas Prometheus.
- Soporte multi-LLM configurable por YAML.
- Publicar paquete `pip install agentic-app-builder`.