Maestría en Inteligencia Artificial Aplicada

Proyecto Integrador

Avance 1: Análisis exploratorio de datos

Profesores:
Dra. Grettel Barceló Alonso
Dr. Luis Eduardo Falcón
Dr. Guillermo Mota Medina

Equipo 59

A01795457	Renzo Antonio Zagni Vestrini
A01362405	Roger Alexei Urrutia Parker
AA01795501	Héctor Raúl Solorio Meneses

Diseño del Sistema: Generación de Aplicaciones con IA Generativa y Agentes

Este documento presenta un diseño técnico ampliado y detallado para un sistema que combina IA generativa, arquitecturas agentic y técnicas de RAG (Retrieval-Augmented Generation) con el fin de permitir que usuarios de negocio creen aplicaciones completas de bases de datos utilizando únicamente prompts en lenguaje natural. El texto combina explicaciones narrativas con bullets técnicos para resaltar puntos clave y facilitar la comprensión desde una perspectiva académica y práctica.

1. Technology Stack

El sistema requiere un conjunto robusto de tecnologías que trabajen de manera integrada para traducir las necesidades expresadas en lenguaje natural en aplicaciones completas.

Backend:

- Lenguajes principales: Python (con frameworks como Flask o FastAPI) o Node.js (con Express).
- ORMs: SQLAlchemy (Python) o Prisma (Node.js) para abstraer la lógica de las bases de datos y generar automáticamente esquemas relacionales.
- Bases de datos: SQLite se usará en fases de prototipado debido a su sencillez y portabilidad, mientras que PostgreSQL será la opción preferida en producción por su robustez y escalabilidad.

Frontend:

- Frameworks modernos: React y Next.js, que permiten construir interfaces modulares, rápidas y escalables.
- Estilos: TailwindCSS y Bootstrap para acelerar la construcción de interfaces atractivas.
- Generación automática de componentes: formularios y tablas creados dinámicamente a partir del esquema generado por el agente de parsing.

Infraestructura:

- Entornos de desarrollo: Replit, ideal para prototipado rápido y pruebas inmediatas.
- Despliegue: servicios como Vercel, Render o configuraciones con Docker Compose, que permiten trasladar rápidamente los prototipos a entornos productivos.

- Validación y calidad de código: pruebas unitarias y de integración con Pytest (Python) o Jest (JavaScript), además de linters como flake8 o ESLint para estandarizar la calidad.

Capa de IA y agentes:

- Modelos de lenguaje: GPT-5 como opción principal, con la posibilidad de incorporar modelos open-source como LLaMA 3 o Falcon para escenarios on-premise o de mayor control.
- Orquestación de agentes: LangChain, AutoGen o Haystack, frameworks que permiten definir flujos agentic complejos donde varios agentes colaboran y validan resultados.
- RAG: módulos de recuperación de información como FAISS, Weaviate o Pinecone, que permiten enriquecer las consultas con información contextual y casos previos.

2. Fuentes de Información

El sistema se alimenta de diferentes fuentes de información que proporcionan contexto, restricciones y ejemplos para guiar la generación automática:

- Prompts de usuarios: son la entrada principal y describen en lenguaje natural la aplicación deseada.
- Bases de conocimiento histórica: ejemplos de aplicaciones desarrolladas previamente en la plataforma de Intelenz, que permiten entrenar y afinar al modelo.
- Esquemas de bases de datos existentes: definen entidades, atributos y relaciones que sirven como referencia para garantizar consistencia y calidad.
- Reglas de negocio: establecen restricciones lógicas y de dominio, fundamentales para asegurar que la aplicación cumpla los requerimientos reales de la organización.
- Documentación técnica de frameworks y librerías: asegura que el código generado esté alineado con las mejores prácticas de la comunidad de desarrollo.

3. RAG (Retrieval-Augmented Generation)

La incorporación de RAG es esencial para reducir la dependencia exclusiva del modelo de lenguaje y mejorar la precisión de los resultados.

- Retriever: busca en bases de conocimiento casos similares, esquemas de referencia y reglas de negocio específicas.

- Augmentation: enriquece el prompt original con la información recuperada, asegurando que el contexto de entrada al modelo sea más rico y detallado.
- Generator: el LLM recibe el prompt enriquecido y produce código estructurado para el backend, el frontend y la base de datos.

Este enfoque minimiza errores, asegura mayor coherencia en la salida y permite que el sistema aprenda dinámicamente de casos pasados.

4. Flujo de Datos

El flujo de datos sigue un pipeline agentic en varias etapas:

1. Entrada: el usuario introduce un prompt en lenguaje natural.
2. Parser Agent: interpreta el prompt y genera un esquema estructurado con entidades, atributos y relaciones.
3. RAG: consulta ejemplos históricos, esquemas previos y reglas de negocio para enriquecer la entrada.
4. Backend Agent: genera modelos ORM, tablas en SQL y endpoints CRUD.
5. Frontend Agent: construye formularios, tablas, dashboards y componentes en React/Next.js.
6. Integrator Agent: ensambla los diferentes módulos en un proyecto completo y valida la consistencia del sistema.
7. Salida: aplicación funcional lista para pruebas iniciales y despliegue.

5. Prompts y Diseño de Interacción

Los usuarios no técnicos interactúan con el sistema mediante prompts en lenguaje natural. Por ejemplo:

- Prompt: *"Necesito una aplicación para gestionar empleados, con nombre, puesto, salario y un dashboard de métricas básicas."*
- Transformación interna:
    - Entidad: Empleado con atributos {nombre, puesto, salario}.
    - Funciones requeridas: CRUD y dashboard.
    - Frameworks recomendados: Flask para backend, React para frontend.

- Base de datos objetivo: PostgreSQL.

El sistema traduce esta descripción en artefactos técnicos listos para ejecución.

6. Generación de Código y SQL

El sistema genera automáticamente:

Modelo en SQLAlchemy:

```
class Empleado(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    nombre = db.Column(db.String(100))
    puesto = db.Column(db.String(100))
    salario = db.Column(db.Float)
```

Tabla en SQL:

```
CREATE TABLE empleados (
  id SERIAL PRIMARY KEY,
  nombre VARCHAR(100),
  puesto VARCHAR(100),
  salario NUMERIC
);
```

Endpoint CRUD en Flask:

```
@app.route('/empleados', methods=['POST'])
def create_empleado():
    data = request.json
    empleado = Empleado(**data)
    db.session.add(empleado)
    db.session.commit()
    return jsonify({"id": empleado.id}), 201
```

Componente en React:

```
export default function EmpleadosPage() {
  return (
    <div>
      <h1>Empleados</h1>
```

```
      <form>{/* Campos: nombre, puesto, salario */}</form>
      <table>{/* Listado de empleados */}</table>
    </div>
  );
}
```

7. Tablas en RDBMS

El diseño de las tablas en un sistema de gestión de bases de datos relacional (RDBMS) es uno de los elementos centrales que el pipeline debe automatizar. A partir del prompt del usuario, el **Parser Agent** extrae las entidades, atributos y relaciones necesarias. Posteriormente, el **Backend Agent** transforma este esquema en código SQL con sentencias DDL (Data Definition Language), que son las encargadas de crear la estructura de la base de datos.

El proceso funciona de la siguiente manera:

1. El LLM interpreta las entidades descritas en lenguaje natural, por ejemplo: *"Necesito una tabla para empleados con nombre, puesto y salario"*.
2. Genera un modelo intermedio en JSON o similar que lista atributos, tipos de datos y restricciones.
3. Con base en ese esquema, construye sentencias SQL DDL como CREATE TABLE, ALTER TABLE y definición de claves primarias y foráneas.
4. Opcionalmente, incorpora índices o validaciones adicionales según las reglas de negocio.

Ejemplo práctico:

- Prompt del usuario: *"Quiero un CRM con leads, contactos y oportunidades"*.
- Esquema generado:
    - Leads: id, nombre, email, fuente.
    - Contactos: id, nombre, empresa, rol, lead_id (relación con Leads).
    - Oportunidades: id, título, monto, estado, contacto_id (relación con Contactos).

DDL generado automáticamente por el LLM:

```
CREATE TABLE leads (
  id SERIAL PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
```

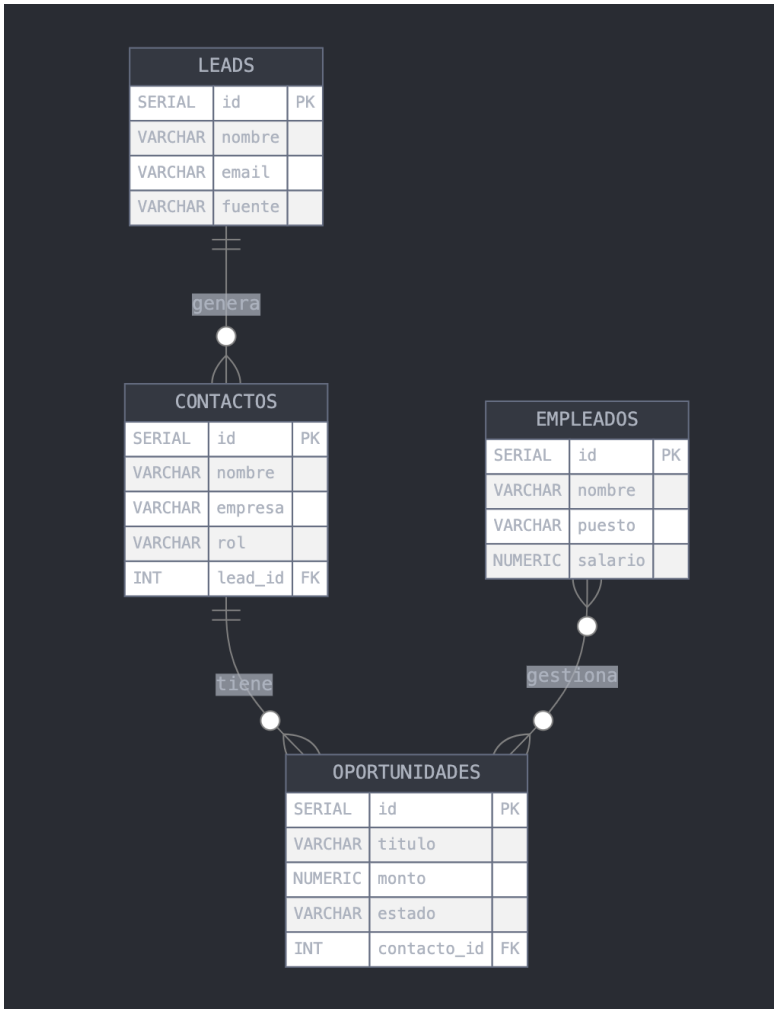
```
  email VARCHAR(100) NOT NULL,
  fuente VARCHAR(100)
);

CREATE TABLE contactos (
  id SERIAL PRIMARY KEY,
  nombre VARCHAR(100) NOT NULL,
  empresa VARCHAR(100),
  rol VARCHAR(100),
  lead_id INT REFERENCES leads(id)
);

CREATE TABLE oportunidades (
  id SERIAL PRIMARY KEY,
  titulo VARCHAR(150) NOT NULL,
  monto NUMERIC,
  estado VARCHAR(50),
  contacto_id INT REFERENCES contactos(id)
);
```

Este enfoque permite que el sistema traduzca directamente una descripción en lenguaje natural en estructuras de base de datos listas para ser implementadas en PostgreSQL o cualquier otro RDBMS compatible. Además, el agente puede generar scripts adicionales para crear índices, vistas o constraints, dependiendo del nivel de detalle solicitado en el prompt o de las reglas de negocio aplicables.

Diagrama ER



Relaciones

- LEADS → CONTACTOS: Un lead puede generar múltiples contactos (1:N)
- CONTACTOS → OPORTUNIDADES: Un contacto puede tener varias oportunidades (1:N)
- EMPLEADOS ↔ OPORTUNIDADES: Relación opcional donde empleados pueden gestionar oportunidades (N:M)

Este diagrama representa perfectamente el flujo de un CRM: desde la captación de leads, pasando por contactos, hasta la gestión de oportunidades de negocio.

8. Retos y Limitaciones Técnicas

A pesar del potencial del sistema, existen desafíos significativos:

- Parsing ambiguo: prompts poco claros pueden generar esquemas incompletos o inconsistentes. Esto requiere mecanismos de validación automática y guías para usuarios.
- Seguridad: el código generado debe protegerse contra vulnerabilidades comunes como inyecciones SQL, ataques XSS o endpoints expuestos sin autenticación.
- Escalabilidad: aunque SQLite es útil en prototipos, es imprescindible migrar a PostgreSQL o soluciones distribuidas en producción.
- Adopción por usuarios no técnicos: las interfaces deben ser intuitivas y guiar al usuario en la construcción de aplicaciones sin conocimiento técnico previo.
- Consistencia de datos: se necesitan validaciones que aseguren que modelos, APIs y UI estén sincronizados.
- Dependencia de LLMs: la precisión depende del corpus de entrenamiento y de ajustes finos. Se deben implementar mecanismos de supervisión.
- Cumplimiento normativo: industrias como salud y fintech requieren trazabilidad, auditorías y privacidad, alineadas con regulaciones como GDPR e HIPAA.

9. Resumen

El diseño propuesto muestra cómo es posible integrar modelos de lenguaje con técnicas de RAG dentro de un pipeline agentic para transformar prompts en aplicaciones completas. El sistema incluye la generación de modelos backend, interfaces frontend, tablas SQL y validación automática. Aunque existen retos importantes relacionados con seguridad, escalabilidad y calidad de generación, la solución plantea un camino innovador para democratizar el desarrollo de aplicaciones y empoderar a usuarios de negocio en la creación de herramientas digitales adaptadas a sus necesidades.