Maestría en Inteligencia Artificial Aplicada

Proyecto Integrador

Avance 2: 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
A01795501	Héctor Raúl Solorio Meneses
A01362405	Roger Alexei Urrutia Parker

Arquitectura de Agentes

Esta arquitectura se basa en un Agente Principal que utiliza un modelo (como GPT-4 o Gemini) para la planificación de alto nivel, y varios Sub-Agentes o Cadenas Especializadas que se enfocan en tareas específicas y utilizan herramientas apropiadas.

1. El Agente Director (Director Agent)

Este agente es el cerebro del sistema. Su función es tomar el prompt de alto nivel del usuario y descomponerlo en un plan de acción secuencial y específico.

- Rol: Planificador, Analista de Requisitos y Coordinador.
- Prompt de Sistema: "Eres un analista de software experto que transforma requisitos en lenguaje natural en un plan de desarrollo estructurado (JSON). Tu objetivo es garantizar la coherencia entre el esquema de la base de datos, el código del ORM, los endpoints del backend y los componentes del frontend."
- Herramientas Clave:
    - StructuredOutputParser: Para forzar la salida inicial a un JSON Pydantic que defina el esquema de la aplicación (entidades, atributos, tipos de datos, operaciones CRUD). Esta es la especificación de diseño que pasará a los agentes trabajadores.
    - TaskDelegationTool: Una herramienta interna que el Agente Director usa para "llamar" o "delegar" tareas a los Agentes Trabajadores.

2. Agentes Trabajadores (Worker Agents)

Cada agente trabajador es un experto en un área de desarrollo y opera de forma autónoma con un conjunto de herramientas especializado.

Agente Trabajador	Especialidad	Tarea Principal	Herramientas Específicas
Agente DB/ORM	Base de Datos y Modelado de Datos.	1. Generar Código SQL (CREATE TABLE). 2. Generar Clases ORM (ej., SQLAlchemy o Mongoose) basadas en el esquema.	SQLValidatorTool: Para comprobar la sintaxis SQL. FileWriteTool: Para guardar archivos del modelo (models.py).
Agente Backend/API	Lógica de Negocio y APIs.	1. Implementar los Controladores/Endpoints CRUD (ej., Flask/Express). 2. Incluir la lógica de serialización y deserialización de datos (ej., Marshmallow o Pydantic).	SyntaxCheckTool: Para autodepurar la sintaxis del lenguaje (backend). EndpointNamingTool: Para verificar convenciones REST.

Agente Trabajador	Especialidad	Tarea Principal	Herramientas Específicas
Agente Frontend/UI	Interfaz de Usuario y Conectividad.	1. Generar componentes de pantallas (Lista, Formulario de Creación/Edición). 2. Implementar la lógica de fetch para interactuar con los endpoints generados por el Agente Backend.	ComponentTemplateTool: Una base de conocimiento interna con templates de React/Vue. FileWriteTool: Para guardar los archivos del frontend (ej., .jsx o .vue).

Flujo de Orquestación Detallado

El proceso sigue un patrón de Planificar Ejecutar Validar Entregar.

1. Requisito Inicial: El usuario introduce: "Crea una app para gestionar películas con título, director y año de estreno."
2. Fase de Planificación (Agente Director):
    - El Director utiliza el StructuredOutputParser para convertir el prompt en un JSON de especificación de diseño (Entidad: Película, Atributos: Título (string), Director (string), Año (int), CRUD: completo).
    - Validación del Diseño: El Director podría invocar un prompt de autocrítica para validar la lógica del diseño antes de delegar.
    - El Director delega la tarea "Generar DB/ORM" al Agente DB/ORM, pasando el JSON de especificación como contexto.
3. Fase de Ejecución (Agente DB/ORM):
    - El Agente DB/ORM genera el SQL y el ORM.
    - El Agente usa la SQLValidatorTool. Si encuentra un error, lo corrige en un nuevo prompt al LLM antes de continuar.
    - Guarda los archivos usando FileWriteTool.
    - Devuelve el código del ORM al Director.
4. Ejecución Paralela/Secuencial (Agente Backend):
    - El Director delega al Agente Backend/API, pasándole el código ORM y el JSON de especificación.
    - El Agente Backend genera los endpoints y usa la SyntaxCheckTool para verificar y corregir su código.
    - Guarda los archivos usando FileWriteTool.
    - Devuelve una lista de URLs de endpoints al Director.
5. Fase de Interfaz (Agente Frontend/UI):
    - El Director delega al Agente Frontend/UI, pasándole el JSON de especificación y la lista de URLs de endpoints.
    - El Agente Frontend genera los componentes y la lógica de conexión.
    - Guarda los archivos usando FileWriteTool.
6. Fase de Ensamblaje y Entrega:

- El Agente Director verifica que todos los archivos requeridos se hayan generado y guardado en la estructura de directorios correcta (ej., app-peliculas/backend/ y app-peliculas/frontend/).
- Entrega el proyecto final al usuario con instrucciones para la ejecución.

Esta arquitectura Agéntica y Modular con LangChain permite la reutilización de agentes y herramientas y garantiza que el código de una capa (frontend) esté correctamente conectado a la capa inmediatamente anterior (backend).

---

1. Agente DB/ORM: Salida y Estructura del Código

El Agente DB/ORM genera dos elementos esenciales: el esquema de la base de datos y la representación en el ORM. La salida debe ser código plano para ser guardado directamente en archivos.

A. Salida de Código (Ejemplo: SQL)

Esta salida se guarda en un archivo como schema.sql y define la estructura de la tabla.

```
SQL
-- Salida: db_code (parte SQL)
-- Archivo: backend/sql/schema.sql
CREATE TABLE books (
    id INT GENERATED ALWAYS AS IDENTITY,
    title VARCHAR(255) NOT NULL,
    author VARCHAR(255) NOT NULL,
    publication_year INT,
    stock INT DEFAULT 0,
    PRIMARY KEY (id)
);
```

B. Salida de Código (Ejemplo: ORM - Python/SQLAlchemy)

Esta salida se guarda en un archivo como models.py y se conecta directamente al SQL anterior. Es la salida más importante para el Agente Backend, ya que define los objetos de datos que usará.

Python

```
# Salida: db_code (parte ORM)
# Archivo: backend/models.py (para Flask/SQLAlchemy)
from sqlalchemy import Column, Integer, String
from sqlalchemy.orm import declarative_base

Base = declarative_base()

class Book(Base):
    __tablename__ = 'books'
    id = Column(Integer, primary_key=True)
    title = Column(String(255), nullable=False)
    author = Column(String(255), nullable=False)
    publication_year = Column(Integer)
    stock = Column(Integer, default=0)

    def to_dict(self):
```

```
        # Función crítica para serializar a JSON
        return {
            "id": self.id,
            "title": self.title,
            "author": self.author,
            "publication_year": self.publication_year,
            "stock": self.stock
        }
```

---

2. Agente Backend/API: Salida de Código y Especificación de la API

El Agente Backend genera dos cosas: el código de los endpoints y una especificación de la API que se convierte en la entrada principal para el Agente Frontend.

A. Especificación de la API (Salida de Datos Estructurados)

Esta salida de datos estructurados (que podría ser una Cadena JSON o una Especificación OpenAPI/Swagger simplificada) es la entrada vital para el Agente Frontend.

Campo	Valor de Ejemplo	Propósito
entity_name	"Book"	Nombre de la entidad.
base_url	"/api/books"	URL base para los endpoints REST.
endpoints	Lista de Objetos	
method	"GET"	HTTP (GET, POST, PUT, DELETE).
url_path	"/"	Ruta completa: /api/books/.
description	"Obtener todos los libros."	
params	[]	Parámetros de la URL (si aplica).
response_schema	{"id": "int", "title": "string", ...}	Esquema esperado de la respuesta.

B. Salida de Código (Ejemplo: Python/Flask API)

Este es el código ejecutable del servidor que implementa la lógica CRUD, utilizando el ORM generado en el paso anterior.

Python
```
# Salida: api_code
# Archivo: backend/app.py (para Flask)
from flask import Flask, jsonify, request
# ... importaciones de ORM ...
from .models import Book # Importa el modelo generado

app = Flask(__name__)
# ... Configuración de base de datos ...

@app.route('/api/books', methods=['GET'])
def list_books():
    books = session.query(Book).all()
    # Utiliza el método to_dict() del ORM
    return jsonify([book.to_dict() for book in books])

@app.route('/api/books', methods=['POST'])
def create_book():
    data = request.json
    new_book = Book(title=data['title'], author=data['author'],
stock=data.get('stock', 0))
    session.add(new_book)
    session.commit()
    return jsonify(new_book.to_dict()), 201

# ... más endpoints para PUT y DELETE ...
```

---

3. Agente Frontend/UI: Salida de Código de las Pantallas

El Agente Frontend utiliza la Especificación de la API como una guía estricta para generar los componentes de la interfaz de usuario que se conectan a los endpoints del backend.

A. Salida de Código (Ejemplo: React Componente de Listado)

Esta salida es un componente completo y autocontenido, guardado como un archivo .jsx o .tsx.

JavaScript
```
// Salida: frontend_code
// Archivo: frontend/src/components/BookList.jsx (para React)
import React, { useState, useEffect } from 'react';
import axios from 'axios';

const BookList = () => {
    const [books, setBooks] = useState([]);
    const [loading, setLoading] = useState(true);

    // Usa la base_url del Agente Backend: /api/books
    const API_URL = "/api/books";

    useEffect(() => {
        axios.get(API_URL)
            .then(response => {
                setBooks(response.data); // Los datos coinciden con el esquema de la
API
```

```
                setLoading(false);
            })
            .catch(error => {
                console.error("Error fetching books:", error);
                setLoading(false);
            });
    }, []);

    if (loading) return <div>Cargando libros...</div>;

    return (
        <div>
            <h2>Inventario de Libros</h2>
            <table>
                <thead>
                    <tr>
                        <th>Título</th>
                        <th>Autor</th>
                        <th>Año</th>
                        <th>Stock</th>
                    </tr>
                </thead>
                <tbody>
                    {books.map(book => (
                        <tr key={book.id}>
                            <td>{book.title}</td>
                            <td>{book.author}</td>
                            <td>{book.publication_year}</td>
                            <td>{book.stock}</td>
                        </tr>
                    ))}
                </tbody>
            </table>
        </div>
    );
};

export default BookList;
```

De esta manera, la orquestación garantiza una comunicación perfecta entre las capas:

1. DB/ORM define el modelo de datos.
2. Backend usa el modelo de datos para construir endpoints coherentes.
3. Frontend usa la especificación de endpoints para construir la UI y las llamadas fetch correctas.

Fundamentación Teórica de los Métodos Aplicados

Durante la fase de preparación de los datos se implementaron técnicas de filtrado y extracción de características con el objetivo de optimizar el rendimiento del modelo de generación automática de aplicaciones, reduciendo la dimensionalidad de los datos, los requerimientos de almacenamiento y el tiempo de entrenamiento. Estas técnicas permiten conservar únicamente las variables más relevantes, eliminando redundancias y ruido que podrían afectar la precisión del modelo.

1. Métodos de Filtrado

Los métodos de filtrado evalúan la relevancia de cada característica de forma independiente del modelo predictivo, basándose en métricas estadísticas:

Umbral de varianza: elimina variables con varianza cercana a cero, ya que no aportan información diferenciadora al modelo. Esto mejora la eficiencia y evita sobreajuste en redes neuronales o modelos de lenguaje como GPT-4.

Correlación: se emplea para detectar y eliminar características altamente correlacionadas entre sí (multicolinealidad). Mantener una sola variable representativa por grupo correlacionado ayuda a simplificar el modelo y reduce la redundancia.

Prueba Chi-cuadrado (χ^2): útil para variables categóricas, mide la dependencia entre cada variable independiente y la variable objetivo. Las características con valores de χ^2 altos son más relevantes para explicar la variabilidad de salida.

ANOVA (Análisis de Varianza): permite identificar si existen diferencias significativas entre las medias de los grupos definidos por una variable categórica. En contextos de clasificación supervisada, ayuda a determinar qué variables influyen significativamente en la predicción.

2. Métodos de Extracción de Características

Estos métodos transforman el conjunto de datos original en un nuevo espacio de menor dimensión conservando la información esencial:

Análisis de Componentes Principales (PCA): técnica estadística que combina variables correlacionadas en componentes ortogonales que explican la mayor parte de la varianza. En el

proyecto, su uso permitiría reducir el número de atributos sin perder representatividad en los datos generados por los sub-agentes.

Análisis Factorial (FA): busca identificar factores latentes que explican las relaciones entre variables observadas. Puede aplicarse para agrupar atributos con comportamientos similares en los datasets generados por los agentes de datos y backend.

Ambos métodos contribuyen a un modelo más generalizable, reduciendo el riesgo de sobreajuste y mejorando la interpretabilidad de los resultados.

Conclusiones de la Fase de "Preparación de los Datos" (CRISP-ML)

En el marco de la metodología CRISP-ML (Cross-Industry Standard Process for Machine Learning), la fase de Data Preparation constituye un paso crítico entre la comprensión de los datos y el modelado.

En este proyecto, las principales conclusiones fueron:

Estandarización y limpieza: se normalizaron estructuras de datos y formatos de entrada utilizados por los agentes (DB/ORM, Backend, Frontend), garantizando coherencia sintáctica y semántica entre las capas del sistema.

Reducción de dimensionalidad: mediante los métodos de filtrado y extracción mencionados, se logró optimizar el conjunto de variables relevantes para la generación automática de código, reduciendo la complejidad sin sacrificar precisión.

Optimización del tiempo de entrenamiento: al eliminar variables redundantes, el modelo principal (basado en LLMs como GPT-4 o Gemini) puede procesar prompts y generar esquemas de aplicación de manera más rápida y eficiente.

Aseguramiento de la calidad de los datos: se implementaron validaciones de consistencia y detección de valores atípicos, esenciales para mantener la estabilidad del sistema de agentes.

Preparación para el modelado: los datos depurados y transformados sirvieron como insumo estructurado para las fases de modelado, evaluación y despliegue, alineándose con los principios de trazabilidad y reproducibilidad propuestos por CRISP-ML.

---

4. Documentación

Resumen de la Arquitectura

Esta aplicación sirve como interfaz frontend para un generador de aplicaciones de base de datos potenciado por IA. Implementa un patrón de asistente de múltiples pasos que guía a los usuarios en el proceso de generar una aplicación full-stack completa.

Jerarquía de Componentes

```
App (page.tsx)
 ├── PromptScreen
 │  ├── Textarea (entrada de prompt)
 │  ├── Input (nombre de entidad)
 │  └── Checkbox (operaciones CRUD)
 ├── MockupEditor
 │  ├── FieldCard (para cada campo)
 │  │  ├── Input (nombre del campo)
 │  │  ├── Input (etiqueta del campo)
 │  │  ├── Select (tipo de campo)
 │  │  └── Checkbox (obligatorio)
 │  └── Button (agregar campo)
 └── ValidationReport
 ├── Badge (indicador de estado)
 ├── Alert (errores/advertencias)
 └── FilePreview (código generado)
```

Gestión de Estado

Estado de la Aplicación

El estado principal se gestiona en `app/page.tsx` usando React hooks:

```
interface Field {
  id: string
  name: string
```

```
  type: string
  label: string
  required: boolean
}

interface AppState {
  currentStep: number
  prompt: string
  entityName: string
  operations: string[]
  schema: { fields: Field[] }
  validationResult: ValidationResult | null
}
```

Flujo de Estado

1. Paso 1 (Prompt): Se captura y almacena la entrada del usuario
2. Paso 2 (Boceto): Se genera el esquema (simulado) y se hace editable
3. Paso 3 (Refinar): El usuario puede modificar el prompt y regenerar
4. Paso 4 (Reporte): Se muestran los resultados finales de la orquestación

Modelos de Datos

Esquema de Campo

```
interface Field {
  id: string          // Identificador único (UUID)
  name: string        // Nombre del campo en la base de datos (ej., "email")
```

```
  type: string        // Tipo de dato (string, number, boolean, etc.)
  label: string       // Etiqueta para mostrar (ej., "Correo Electrónico")
  required: boolean   // Indicador de validación
}
```

Resultado de Validación

```
interface ValidationResult {
  status: 'success' | 'warning' | 'error'
  validations: {
    errors: string[]
    warnings: string[]
  }
  generatedFiles: GeneratedFile[]
  metrics: {
    totalFiles: number
    linesOfCode: number
    generationTime: string
  }
}

interface GeneratedFile {
  name: string        // Nombre del archivo (ej., "user.model.ts")
  type: string        // Tipo de archivo (model, api, component)
  content: string     // Contenido del archivo
  size: string        // Tamaño del archivo (ej., "2.4 KB")
}
```

Documentación de Componentes

Componente PromptScreen

Propósito: Captura los requisitos y configuración del usuario

Props:

```
interface PromptScreenProps {
  prompt: string
  setPrompt: (value: string) => void
  entityName: string
  setEntityName: (value: string) => void
  operations: string[]
  setOperations: (value: string[]) => void
  onNext: () => void
}
```

Características:

- Entrada de prompt con texto enriquecido y contador de caracteres
- Validación del nombre de la entidad
- Selección múltiple de operaciones CRUD
- Validación de formulario antes de continuar

Componente MockupEditor

Propósito: Muestra y permite editar el esquema generado

Props:

```
interface MockupEditorProps {
  schema: { fields: Field[] }
```

```
  setSchema: (schema: { fields: Field[] }) => void
  onBack: () => void
  onRefine: () => void
  onConfirm: () => void
}
```

Características:

- Lista de campos con edición en línea
- Agregar/eliminar campos
- Reordenamiento drag-and-drop (mejora futura)
- Selección de tipo con íconos
- Alternar obligatoriedad del campo

Tipos de Campos Soportados:

- string (entrada de texto)
- number (entrada numérica)
- boolean (checkbox)
- date (selector de fecha)
- email (validación de correo)
- url (validación de URL)
- text (textarea)
- select (desplegable)

Componente ValidationReport

Propósito: Muestra resultados de orquestación y código generado

Props:

```
interface ValidationReportProps {
  result: ValidationResult
  onBack: () => void
  onStartNew: () => void
}
```

Características:

- Badge de estado (success/warning/error)
- Listas de errores y advertencias
- Vista previa de archivos generados con resaltado de sintaxis
- Visualización de métricas
- Funcionalidad de descarga (mejora futura)

Sistema de Estilos

Tokens de Diseño

Definidos en app/globals.css:

```
@theme inline {
  /* Colores */
  --color-background: #0a0a0a;
  --color-foreground: #ededed;
  --color-primary: #06b6d4;
  --color-accent: #3b82f6;
  --color-muted: #262626;
```

```
  --color-border: #27272a;

  /* Tipografía */
  --font-sans: var(--font-geist-sans);
  --font-mono: var(--font-geist-mono);

  /* Espaciado */
  --radius: 0.5rem;
}
```

Clases Utilitarias

Utilidades personalizadas para estilo consistente:

- .step-indicator - Estilo del indicador de progreso
- .field-card - Estilo de tarjeta de campo del esquema
- .code-preview - Estilo de bloque de código con resaltado de sintaxis

Puntos de Integración API

Endpoint Generar Esquema

Endpoint: POST /api/generate-schema

Propósito: Convierte el prompt en lenguaje natural a un esquema estructurado

Implementación Actual: Simulada con datos mock

Implementación Futura:

- Llamar al Director Agent para analizar el prompt
- Invocar DB/ORM Agent para generar esquema

- Devolver definiciones estructuradas de campos

Endpoint de Orquestación

Endpoint: POST /api/orchestrate

Propósito: Dispara la generación completa de la aplicación

Implementación Actual: Simulada con resultados de validación mock

Implementación Futura:

- Validar estructura del esquema
- Coordinar todos los Worker Agents (DB, Backend, Frontend)
- Generar código completo de la aplicación
- Ejecutar validaciones y pruebas
- Devolver informe completo

Consideraciones de Rendimiento

Estrategias de Optimización

1. Memoización de Componentes: Usar React.memo para tarjetas de campo y evitar re-renderizados innecesarios
2. Carga Diferida: Componentes de vista previa de código cargados de forma lazy
3. Debounce: Campos de entrada usan debounce para reducir actualizaciones de estado
4. Scrolling Virtual: Para esquemas grandes con muchos campos (mejora futura)

Tamaño del Bundle

- Splitting automático de código en Next.js

- Imports dinámicos para componentes pesados
- Tree-shaking para componentes UI no usados

Consideraciones de Seguridad

Validación de Entrada

- Sanitizar prompts antes de enviar al backend
- Validar nombres de entidad (alfanuméricos, sin caracteres especiales)
- Limitar longitud del prompt para evitar abusos
- Validar nombres de campo contra patrones de inyección SQL

Seguridad API

- Implementar rate limiting en endpoints de generación
- Añadir autenticación para producción
- Validar estructura del esquema antes de la orquestación
- Sanitizar código generado antes de mostrar

Estrategia de Pruebas

Pruebas Unitarias

- Test de renderizado de componentes
- Test de lógica de gestión de estado
- Test de funciones de validación
- Manejo de respuestas mock de API

Pruebas de Integración

- Test de flujo multi-paso
- Envío de formularios y validación
- Operaciones de edición de esquema
- Escenarios de manejo de errores

Pruebas E2E

- Recorrido completo de usuario desde prompt hasta reporte
- Casos límite y escenarios de error
- Compatibilidad en navegadores
- Pruebas de diseño responsivo

Despliegue

Variables de Entorno

```
# Configuración API
NEXT_PUBLIC_API_URL=https://api.example.com

# Feature Flags
NEXT_PUBLIC_ENABLE_REFINEMENT=true
NEXT_PUBLIC_ENABLE_DOWNLOAD=true
```

Configuración de Build

```
{
  "scripts": {
```

```
    "dev": "next dev",
    "build": "next build",
    "start": "next start",
    "lint": "next lint"
  }
}
```

Plataformas de Despliegue

- Vercel: Recomendado (soporte nativo Next.js)

Mejoras Futuras

Funciones Planeadas

1. Colaboración en tiempo real: Múltiples usuarios editando el mismo esquema
2. Historial de versiones: Seguimiento de iteraciones con vista de diferencias
3. Biblioteca de plantillas: Esquemas predefinidos para casos comunes
4. Opciones de exportación: SQL, Prisma, TypeORM, etc.
5. Sugerencias de IA: Recomendaciones de campos en tiempo real
6. Reglas de validación: Configuración avanzada de validación de campos
7. Relaciones: Definir llaves foráneas y relaciones entre entidades
8. Modo de previsualización: Vista previa en vivo de componentes UI generados

Deuda Técnica

- Añadir límites de error completos

- Implementar estados de carga adecuados
- Mejorar accesibilidad (ARIA, navegación por teclado)
- Optimizar re-renderizados con mejor gestión de estado
- Tipos TypeScript completos para todos los componentes

Resolución de Problemas

Problemas Comunes

Problema: El esquema no se genera

- Solución: Revisar la consola por errores de API, verificar que el prompt no esté vacío

Problema: Los campos no guardan ediciones

- Solución: Asegurarse de que los nombres de campo sean únicos, verificar errores de validación

Problema: El reporte de validación no se muestra

- Solución: Verificar formato de respuesta del endpoint de orquestación

Guías de Contribución

Estilo de Código

- Usar TypeScript para todos los nuevos componentes
- Seguir configuración ESLint
- Usar Prettier para formato
- Escribir mensajes de commit descriptivos

Directrices de Componentes

- Mantener componentes pequeños y enfocados
- Usar composición sobre herencia
- Implementar tipos de props adecuados
- Añadir comentarios JSDoc para lógica compleja

Proceso de Pull Request

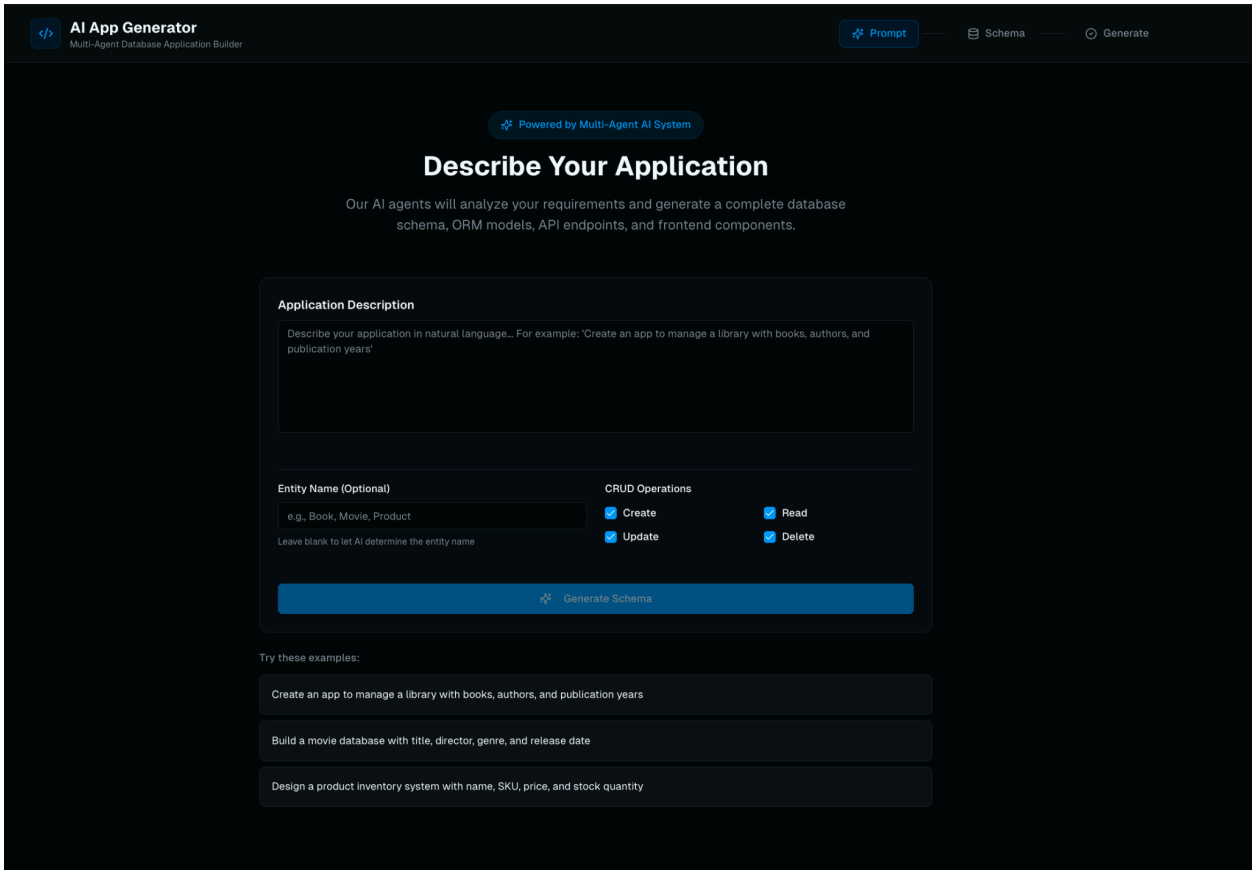
1. Crear branch de feature desde main
2. Implementar cambios con tests
3. Actualizar documentación
4. Enviar PR con descripción
5. Atender comentarios de revisión
6. Merge después de aprobación

Recursos

- Documentación Next.js
- Tailwind CSS
- shadcn/ui
- Manual de TypeScript

---

5. Imágen de la App



(La aplicación aceptara mutiples idiomas y se agregaremos la funcionalidad para cambiarle el idioma dentro de la misma app)