

# Postgres MCP + LangGraph Agent System

---

Un sistema completo de asistencia para bases de datos PostgreSQL que presenta arquitecturas duales de agentes: herramientas directas de LangGraph e integración con servidor MCP (Model Context Protocol). Incluye una interfaz de chat en Streamlit con características avanzadas como seguimiento de tiempos de procesamiento, gestión de salidas de herramientas y exportación de conversaciones.

## Visión General de la Arquitectura

El sistema consiste en múltiples componentes interconectados que trabajan juntos para proporcionar interacción inteligente con la base de datos:

### Componentes Principales

- **Capa de Base de Datos** (`db.py`) - Gestión asíncrona de conexiones y operaciones PostgreSQL
- **Servidor MCP** (`server4.py`) - Servidor MCP basado en HTTP que expone herramientas de base de datos
- **Capa de Agentes** - Dos implementaciones de agentes para diferentes patrones de interacción
- **Interfaz de Usuario** (`streamlit_chat.py`) - Interfaz de chat moderna con características avanzadas

### Flujo de Datos

1. **Entrada del Usuario** → La UI de Streamlit recoge mensajes y configuración
2. **Selección de Agente** → Enruta al agente directo LangGraph o al agente cliente MCP
3. **Ejecución de Herramientas** → El agente directo llama herramientas en proceso; el agente MCP usa transporte HTTP
4. **Operaciones de Base de Datos** → Las herramientas delegan al pool de conexiones asyncpg en `db.py`
5. **Procesamiento de Respuesta** → Los resultados se serializan y retornan a través de la cadena de mensajes
6. **Visualización en UI** → Mensajes formateados con marcas de tiempo, tiempos de procesamiento y salidas de herramientas

## Estructura de Archivos y Componentes

### Scripts Python Principales

#### `db.py` - Capa de Operaciones de Base de Datos

**Propósito:** Interfaz asíncrona de base de datos PostgreSQL con pooling de conexiones y operaciones de metadatos.

#### Funciones Clave:

- `get_pool()` / `close_pool()` - Gestión del ciclo de vida del pool de conexiones
- `list_tables(schema)` - Enumera tablas con metadatos (nombre, schema, conteos de filas)
- `describe_table(table_name, schema)` - Información de columnas, tipos de datos, restricciones y estadísticas

- `get_table_sample(table_name, limit, schema)` - Recupera filas de muestra con límites configurables
- `execute_sql(query)` - Ejecuta consultas SQL arbitrarias con formato de resultados

**Características:**

- Pooling de conexiones AsyncPG con tamaños min/max configurables
- Soporte para nombres de tabla calificados por schema
- Extracción completa de metadatos de columnas
- Estimación de conteo de filas para tablas grandes
- Serialización de resultados en formato JSON
- Manejo completo de errores y logging

**server4.py - Implementación del Servidor MCP**

**Propósito:** Servidor HTTP basado en FastMCP que expone herramientas de PostgreSQL mediante Model Context Protocol.

**Herramientas Expuestas:**

- `list_tables(schema)` - Lista las tablas disponibles con metadatos
- `describe_table(table_name, schema)` - Información detallada de la estructura de la tabla
- `get_table_sample(table_name, limit, schema)` - Muestra de datos de la tabla
- `execute_sql(query)` - Ejecuta consultas SQL

**Características:**

- Transporte SSE (Server-Sent Events) para respuestas streaming
- Comunicación basada en sesiones con limpieza automática
- Endpoint de health check (`/health`)
- Documentación completa de herramientas con descripciones de parámetros
- Manejo asíncrono de solicitudes con respuestas de error apropiadas

**agent\_langchain.py - Agente Directo LangGraph**

**Propósito:** Agente ReAct de LangGraph con integración directa de herramientas (sin overhead de red).

**Arquitectura:**

- Definiciones de herramientas LangChain con validación Pydantic
- System prompt con instrucciones específicas para PostgreSQL
- Gestión de historial de mensajes para contexto de conversación
- Procesamiento de resultados de herramientas y manejo de errores

**Características:**

- Llamadas directas a funciones de herramientas de base de datos (sin overhead HTTP)
- System prompt completo con guías de uso
- Preservación del historial de conversación
- Manejo estructurado de errores y logging

### agent\_mcp\_client.py - Agente Cliente MCP

**Propósito:** Agente LangGraph que descubre y usa herramientas de servidores MCP sobre HTTP.

**Arquitectura:**

- Descubrimiento dinámico de herramientas desde servidor MCP
- Wrapping automático de herramientas para compatibilidad con LangChain
- Cliente SSE para comunicación en tiempo real
- Mapeo de parámetros de herramientas consciente de schema

**Características:**

- Descubrimiento y caché de herramientas en tiempo de ejecución
- Manejo de herramientas estructuradas vs entrada única
- Manejo completo de errores y logging
- Procesamiento y serialización de resultados de herramientas

### streamlit\_chat.py - Interfaz de Usuario de Chat

**Propósito:** Interfaz de chat moderna y rica en características para interacción con base de datos.

**Características Clave:**

- **Soporte Dual de Agentes:** Alternar entre modos de agente Directo y MCP
- **Controles Avanzados de UI:**
  - Botones de New Chat / Clear Chat
  - Toggle de visibilidad de salida de herramientas
  - Visualización de tiempo de procesamiento con codificación por color
  - Exportación de conversación a JSON
- **Visualización de Mensajes:**
  - Orden cronológico inverso (más reciente primero)
  - Burbujas de mensajes codificadas por color (usuario=azul, asistente=púrpura, herramientas=naranja)
  - Marcas de tiempo e indicadores de tiempo de procesamiento
  - Botones de copia para contenido de mensajes
  - Secciones expandibles de salida de herramientas
- **Interfaz de Entrada:**
  - Área de texto grande para consultas multilínea
  - Envío basado en formulario con validación
  - Sugerencias de consultas de ejemplo para nuevas conversaciones

### smoke\_mcp.py - Testing del Servidor MCP

**Propósito:** Suite completa de pruebas de humo para funcionalidad del servidor MCP.

**Cobertura de Pruebas:**

- Inicialización del servidor y detección de capacidades
- Descubrimiento de herramientas y validación de metadatos

- Ejecución básica de herramientas (list\_tables)
- Operaciones avanzadas (describe\_table, get\_table\_sample)
- Manejo de errores y parsing de respuestas

#### Características:

- Ejecución automatizada de pruebas con salida clara
- Parsing y validación de resultados JSON
- Manejo elegante de datos faltantes
- Logging completo para debugging

#### `__init__.py` - Inicialización de Paquete

**Propósito:** Archivo marcador de paquete Python (actualmente vacío).

## Configuración

### Variables de Entorno

Ver `.env.example` para una plantilla de configuración completa. Las variables clave incluyen:

#### Configuración de Base de Datos

```
PGHOST=localhost           # Host de PostgreSQL
PGPORT=5432                # Puerto de PostgreSQL
PGUSER=your_username      # Usuario de base de datos
PGPASSWORD=your_password  # Contraseña de base de datos
PGDATABASE=your_database  # Nombre de base de datos
PGSSL=false                # Modo SSL (false/require)
PGPOOL_MIN_SIZE=1         # Tamaño mínimo del pool de conexiones
PGPOOL_MAX_SIZE=10        # Tamaño máximo del pool de conexiones
PGPOOL_COMMAND_TIMEOUT=30 # Timeout de consulta en segundos
```

#### Configuración del Servidor

```
POSTGRES_MCP_HOST=0.0.0.0 # Dirección de enlace del servidor MCP
POSTGRES_MCP_PORT=8010    # Puerto del servidor MCP
POSTGRES_MCP_PATH=/mcp    # Ruta del endpoint MCP
POSTGRES_MCP_URL=http://localhost:8010/mcp # URL completa del servidor MCP
```

#### Configuración AI/ML

```
OPENAI_API_KEY=your_api_key # Clave API de OpenAI
OPENAI_MODEL=gpt-4o-mini    # Modelo por defecto para chat
```

## Inicio Rápido

### 1. Instalar Dependencias

```
cd postgres_gpt
pip install -r requirements.txt
```

### 2. Configurar Entorno

```
# Copiar y editar variables de entorno
cp .env.example .env
# Editar .env con tus credenciales de base de datos y claves API
```

### 3. Iniciar Servidor MCP

```
python -m postgres_gpt.server4
# El servidor se ejecuta en http://localhost:8010
# Health check: curl http://localhost:8010/health
```

### 4. Probar Servidor MCP (Opcional)

```
python -m postgres_gpt.smoke_mcp
# Ejecuta pruebas de humo completas
```

### 5. Lanzar Interfaz de Chat

```
streamlit run postgres_gpt/streamlit_chat.py
# Abre el navegador en http://localhost:8501
```

## Ejemplos de Uso

### Exploración Básica de Tablas

```
Usuario: Lista todas las tablas en la base de datos
Asistente: Aquí están las tablas disponibles:
- public.actor
- public.address
- public.category
...
```

Usuario: Describe la tabla actor  
Asistente: La tabla public.actor tiene la siguiente estructura:

- actor\_id (integer, primary key, auto-increment)
- first\_name (varchar(45), not null)
- last\_name (varchar(45), not null)
- last\_update (timestamp, not null)

Total de filas: 200

## Consultas Complejas

Usuario: Muéstrame los 3 actores principales por cantidad de películas  
Asistente: Necesitaré unir las tablas actor y film\_actor...

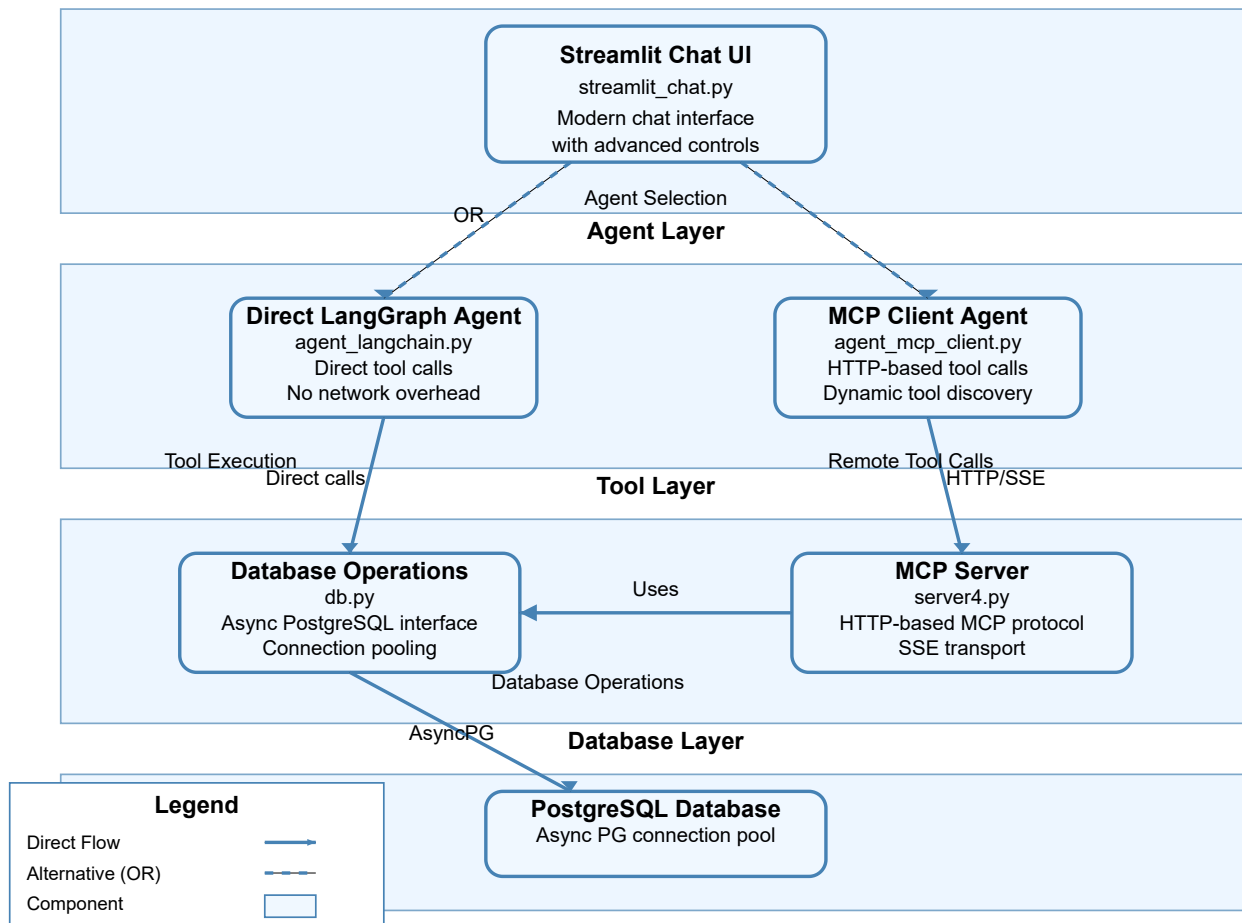
[Ejecución de herramienta con consulta SQL]  
SELECT a.first\_name, a.last\_name, COUNT(fa.film\_id) as movie\_count  
FROM actor a  
JOIN film\_actor fa ON a.actor\_id = fa.actor\_id  
GROUP BY a.actor\_id, a.first\_name, a.last\_name  
ORDER BY movie\_count DESC  
LIMIT 3;

Resultados:

1. Gina Degeneres - 42 películas
2. Walter Torn - 41 películas
3. Mary Keitel - 40 películas

## Diagrama de Arquitectura

## Postgres MCP + LangGraph Agent System Architecture User Interface Layer



El diagrama muestra la arquitectura de cuatro capas:

- **Capa de Interfaz de Usuario:** Interfaz de chat basada en Streamlit con controles avanzados
- **Capa de Agentes:** Implementaciones duales de agentes (Directo y cliente MCP)
- **Capa de Herramientas:** Herramientas de operación de base de datos con diferentes rutas de ejecución
- **Capa de Base de Datos:** PostgreSQL con pooling de conexiones asíncrono

Los datos fluyen desde la entrada del usuario a través de la selección de agente, ejecución de herramientas y operaciones de base de datos, con las respuestas fluyendo de vuelta a través de las capas.

## Streamable HTTP vs SSE

Streamable HTTP es un patrón de respuesta que mantiene el modelo simple de request/response HTTP pero transmite chunks tan pronto como están listos. Típicamente usa **Transfer-Encoding: chunked** (o frames de datos HTTP/2) para entregar payloads parciales más un resumen final sin cambiar de protocolo.

Por qué a menudo supera a SSE

- Funciona sobre stacks HTTP estándar (CDNs, proxies, load balancers) sin requerir upgrades a **text/event-stream**.
- Lleva payloads binarios o mixtos (JSON, embeddings, archivos) sin el framing de solo texto de SSE.
- Se adapta mejor a respuestas multi-parte: tokens tempranos primero, metadatos estructurados o estadísticas de costo al final.

- Manejo simple del cliente: cualquier cliente HTTP que soporte streaming de cuerpos puede consumirlo; no se necesita parsing de eventos.

## Dónde SSE aún puede ganar

- Semántica de reintento incorporada con `Last-Event-ID` si necesitas reconexión automática.
- Muy ligero para feeds de eventos pequeños y solo texto.
- Amplio soporte de navegador con `EventSource` cuando no controlas los clientes.

## Guía Práctica

- Usa streamable HTTP cuando necesites compatibilidad end-to-end a través de proxies/CDNs, frames binarios, o salidas estructuradas multi-parte (ej., deltas + resumen JSON final) mientras te mantienes en HTTP vanilla.
- Quédate con SSE para actualizaciones en vivo simples y solo texto donde `EventSource` es suficiente y la lógica de reconexión importa más que la flexibilidad del payload.

Referencia: comparación y benchmarks en [Medium: Streamable HTTP vs SSE](#).

## Monitoreo y Debugging

### Logging

- Todos los componentes escriben a `agent_activity.log`
- Logging completo de request/response
- Timing de ejecución de herramientas y errores
- Estado del pool de conexiones de base de datos

### Health Checks

- Servidor MCP: `GET /health` retorna estado de conectividad de base de datos
- UI Streamlit: Feedback visual para respuestas del agente
- Validación de herramientas: Manejo automático de errores y reintentos

### Monitoreo de Rendimiento

- Tiempos de ejecución de consultas rastreados en UI
- Logging de utilización del pool de conexiones
- Timing de pasos de razonamiento del agente

## Consideraciones de Seguridad

### Seguridad de Base de Datos

- Usar conexiones SSL para producción (`PGSSL=require`)
- Implementar autenticación y autorización apropiadas
- Evitar exponer la herramienta `execute_sql` en entornos no confiables
- Usar consultas parametrizadas cuando sea posible

### Seguridad de API

- Protección de clave API de OpenAI mediante variables de entorno
- Validación de entrada en todo SQL proporcionado por usuario
- Consideraciones de rate limiting para despliegue en producción

## Seguridad de Red

- Enlazar servidor MCP a interfaces apropiadas
- Usar HTTPS en entornos de producción
- Implementar políticas CORS apropiadas si es necesario

## Opciones de Despliegue

### Desarrollo Local

- Máquina única con PostgreSQL local
- Acceso directo a herramientas para máximo rendimiento
- Capacidades completas de debugging y logging

### Despliegue Containerizado

```
FROM python:3.11-slim
WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt
COPY . .
EXPOSE 8010 8501
CMD ["python", "-m", "postgres_gpt.server4"]
```

### Despliegue en Cloud

- Instancias separadas de servidor MCP y UI
- Load balancer para alta disponibilidad
- Servicios PostgreSQL gestionados (RDS, Cloud SQL, etc.)
- API Gateway para control de acceso externo

## Contribuyendo

### Setup de Desarrollo

1. Fork del repositorio
2. Crear rama de feature
3. Hacer cambios con pruebas completas
4. Actualizar documentación
5. Enviar pull request

### Testing

- Ejecutar pruebas de humo: `python -m postgres_gpt.smoke_mcp`






- Probar ambos modos de agente en UI Streamlit
- Verificar operaciones de base de datos con varios schemas
- Comprobar manejo de errores y casos edge

## Estándares de Código

- Type hints para todos los parámetros de función
- Docstrings completos
- Async/await para operaciones de base de datos
- Logging estructurado con niveles apropiados
- Manejo de errores con mensajes significativos

## Changelog

### Actualizaciones Recientes (v2.0)

-  **UI de Chat Mejorada:** Rediseño completo con estilo moderno, seguimiento de tiempos de procesamiento y controles avanzados
-  **Respuestas de Agente Mejoradas:** Eliminada repetición de system prompt para salida más limpia
-  **Mejor Monitoreo:** Logging completo y seguimiento de rendimiento
-  **Refinamientos de Arquitectura:** Separación más limpia entre modos de agente directo y MCP
-  **Documentación:** README completo con diagramas de arquitectura y ejemplos de uso

### Versiones Previas

- v1.0: Implementación inicial con servidor MCP básico y UI Streamlit
- v0.5: Integración de agente directo LangGraph
- v0.1: Operaciones principales de base de datos y definiciones de herramientas

---

**Construido con:** Python 3.10+, FastMCP, LangGraph, Streamlit, asyncpg **Licencia:** MIT