Análisis y Diseño de Arquitectura para "Analizador de Complejidades"

¡Excelente iniciativa de proyecto! La combinación de análisis algorítmico clásico con la potencia de los LLMs es innovadora y muy relevante. A continuación, presento el diseño de arquitectura y los componentes solicitados.

---

1. Diseño de Arquitectura de Software

Propongo una **arquitectura en capas (N-Tier)**, que separa las responsabilidades del sistema, facilitando su mantenimiento, escalabilidad y desarrollo modular.

- **Capa de Presentación (Frontend):**
  - **Responsabilidad:** Interactuar con el usuario. Permitirá ingresar el pseudocódigo, iniciar el análisis, visualizar los resultados de forma interactiva y exportar los reportes.
  - **Componentes:** Podría ser una aplicación web (Single Page Application - SPA) o una interfaz de escritorio simple.
- **Capa de Lógica de Negocio (Backend - Core):**
  - **Responsabilidad:** Orquestar todo el proceso de análisis. Es el corazón del sistema.
  - **Componentes:**
    - **Servicio de Orquestación:** Recibe las solicitudes de la capa de presentación.
    - **Módulo de Parsing y Análisis Sintáctico:** Interpreta el pseudocódigo de entrada.
    - **Módulo de Análisis de Complejidad:** Aplica las reglas matemáticas para calcular O, Ω y Θ.
    - **Módulo de Generación de Reportes:** Estructura la salida del análisis.
- **Capa de Integración (Servicios Externos):**
  - **Responsabilidad:** Comunicarse con servicios de terceros, en este caso, los Modelos de Lenguaje (LLMs).
  - **Componentes:** Un **Cliente de API para LLMs** que gestiona las solicitudes (traducción, validación, clasificación) y maneja las respuestas.
- **Capa de Persistencia (Base de Datos):**
  - **Responsabilidad:** Almacenar datos relevantes como el historial de análisis, los algoritmos ingresados, los resultados y, potencialmente, una base de conocimiento de patrones algorítmicos.
  - **Componentes:** Un sistema de gestión de base de datos (SQL o NoSQL, según la necesidad de estructurar las relaciones).

---

2. Módulos y Componentes Detallados

Módulo	Responsabilidad	Entradas	Salidas	Dependencias
Parser	Validar la sintaxis del pseudocódigo y generar un Árbol de Sintaxis Abstracta (AST).	String con pseudocódigo.	Árbol AST.	Gramática definida.
Analizador de Complejidades	Recorrer el AST para identificar estructuras (ciclos, recursiones) y calcular la complejidad.	Árbol AST.	Objeto Complejidad.	-
LLM Service	Interactuar con la API de un LLM para asistir en tareas de NLP y validación.	Texto (lenguaje natural o pseudocódigo).	Texto procesado (pseudocódigo, validación, etc.).	API del LLM.
Generador de Reportes	Consolidar toda la información del análisis en un formato estructurado.	Objeto Algoritmo, Objeto Complejidad.	Archivo PDF o JSON.	-

API/Controlador	Exponer los servicios del sistema a través de endpoints para que la capa de presentación los consuma.	Peticiones HTTP.	Respuestas HTTP (JSON).	Todos los módulos de negocio.

---

3. Diagrama UML de Clases

Este diagrama representa las entidades principales del sistema y sus relaciones, basado en tu propuesta y refinado para mayor detalle.

Fragmento de código

```
classDiagram
    class Usuario {
        +String nombre
        +String rol
        +ingresarAlgoritmo(codigo: String): Algoritmo
        +solicitarAnalisis(algoritmo: Algoritmo): Reporte
    }

    class Algoritmo {
        +String id
        +String codigoFuente
        +TipoAlgoritmo tipo
        +AST arbolSintactico
    }

    enum TipoAlgoritmo {
        RECURSIVO
        ITERATIVO
    }
```

```
class Parser {
    -Grammar gramatica
    +parsear(codigo: String): AST
    +validarSintaxis(codigo: String): Boolean
}

class AnalizadorComplejidades {
    +calcularO(ast: AST): String
    +calcularOmega(ast: AST): String
    +calcularTheta(ast: AST): String
    +generarJustificacion(ast: AST): String
    +analizar(algoritmo: Algoritmo): Complejidad
}

class Complejidad {
    +String notacionO
    +String notacionOmega
    +String notacionTheta
    +String justificacionMatematica
}

class LLMService {
    -String apiKey
    -String modelo
    +traducirNaturalAPseudocodigo(texto: String): String
    +validarAnalisis(complejidad: Complejidad): String
    +clasificarPatron(algoritmo: Algoritmo): String
}

class Reporte {
    +String id
    +Algoritmo algoritmoAnalizado
    +Complejidad resultadoComplejidad
    +String validacionLLM
    +exportarPDF(): File
    +exportarJSON(): File
}

Usuario --> Algoritmo : "ingresa"
Usuario --> AnalizadorComplejidades : "solicita análisis"
AnalizadorComplejidades --> Algoritmo : "utiliza"
AnalizadorComplejidades --> Parser : "utiliza"
AnalizadorComplejidades --> LLMService : "puede utilizar para validar"
```

```
    AnalizadorComplejidades --> Complejidad : "genera"
    AnalizadorComplejidades --> Reporte : "genera"
```

---

4. Diagrama UML de Casos de Uso

Muestra las interacciones clave que un usuario (en este caso, un "Estudiante/Investigador") puede tener con el sistema.

Fragmento de código

```
graph TD
    subgraph "Sistema Analizador de Complejidades"
        UC1("Ingresar Pseudocódigo")
        UC2("Generar Análisis de Complejidad")
        UC3("Asistir con LLM para traducción")
        UC4("Validar Análisis con LLM")
        UC5("Exportar Reporte")
    end

    Usuario --|> EstudianteInvestigador
    EstudianteInvestigador -- (UC1)
    EstudianteInvestigador -- (UC2)
    EstudianteInvestigador -- (UC5)

    UC2 ..> UC3 : "include"
    UC2 ..> UC4 : "extend"
```

---

5. Flujo UML de Actividades

Este diagrama detalla el proceso paso a paso desde que el usuario introduce un algoritmo hasta que recibe el reporte final.

Fragmento de código

```
graph TD
    A[Inicio] --> B{Ingresar Pseudocódigo};
    B --> C{¿Sintaxis válida?};
    C -- No --> D[Mostrar Error de Sintaxis];
    D --> E[Fin];
    C -- Sí --> F[Generar Árbol de Sintaxis Abstracta (AST)];
    F --> G[Analizar AST y Calcular Complejidades O, Ω, Θ];
    G --> H{¿Validar con LLM?};
    H -- Sí --> I[Enviar análisis a LLM para obtener feedback];
    I --> J[Incorporar feedback del LLM al reporte];
    J --> K[Generar Reporte Detallado];
    H -- No --> K;
    K --> L{¿Exportar Reporte?};
    L -- Sí --> M[Generar Archivo (PDF/JSON)];
    M --> E;
    L -- No --> E;
```

---

6. Tecnologías Recomendadas (Stack en Python)

- **Backend (API): FastAPI** o **Flask**. FastAPI es moderno, rápido y tiene generación automática de documentación, lo cual es ideal.
- **Parsing y Análisis de Gramática:**
  - **Lark:** Una librería moderna para parsing que permite definir gramáticas en EBNF. Es muy potente y relativamente fácil de usar.
  - **PLY (Python Lex-Yacc):** La implementación clásica de Lex y Yacc en Python. Es muy robusta, aunque con una curva de aprendizaje un poco más alta.
- **Integración con LLMs:**
  - **OpenAI API Client (openai):** Si usas modelos de OpenAI.
  - **Google AI Generative Language (google-generativeai):** Para la familia de modelos Gemini.
  - **Hugging Face transformers y requests:** Para interactuar con una gran variedad de modelos.
- **Generación de PDF: ReportLab** o **WeasyPrint**.

- **Base de Datos: PostgreSQL** con **SQLAlchemy** como ORM para una solución robusta, o **MongoDB** con **Pymongo** si prefieres un enfoque NoSQL más flexible para almacenar los reportes.