

A blue parallelogram and a light green parallelogram are positioned in the upper-left corner of the slide. The blue shape is partially behind the green one. Both shapes are oriented diagonally, with their longer sides running from the top-left towards the bottom-right.

Proyecto Final: DataQuery IA



Contexto

En el ecosistema actual de análisis de datos, existe una barrera técnica significativa entre los usuarios empresariales (no técnicos) y los datos almacenados en bases de datos SQL. Los equipos de negocio necesitan respuestas rápidas pero dependen de:

- Desarrolladores para escribir consultas SQL
- Analistas de datos para crear dashboards
- Ciclos largos de solicitud → desarrollo → entrega



Objetivo

Transformar la manera en que las organizaciones interactúan con sus datos, convirtiendo preguntas naturales en insights accionables mediante inteligencia artificial, visualizaciones automáticas y seguridad enterprise, todo en un flujo unificado y accesible para usuarios no técnicos

Automatización Inteligente de Flujos

- Automatizar el proceso completo: pregunta → SQL → ejecución → visualización
- Reducir tiempo de respuesta de horas/días a segundos
- Eliminar dependencia de equipos técnicos para consultas rutinarias



Visualización Automática e Inteligente

- Detectar automáticamente cuándo una consulta necesita visualización
- Seleccionar el tipo de gráfico más apropiado (barras, líneas, pie, etc.)
- Generar visualizaciones sin intervención manual

Experiencia de Usuario Fluida

- Experiencia de Usuario Fluida
- Interfaz conversacional natural en español
- Historial de conversaciones persistente
- Respuestas enriquecidas (datos + gráficos + metadata)



Stack Tecnológico

Backend principal:

- FastAPI - Framework web moderno
- AWS Bedrock - Servicio de LLM
- PostgreSQL - Base de datos principal
- DistilBERT - Modelo de detección de gráficos
- Matplotlib/Seaborn - Generación de visualizaciones

Servicio de Persistencia:

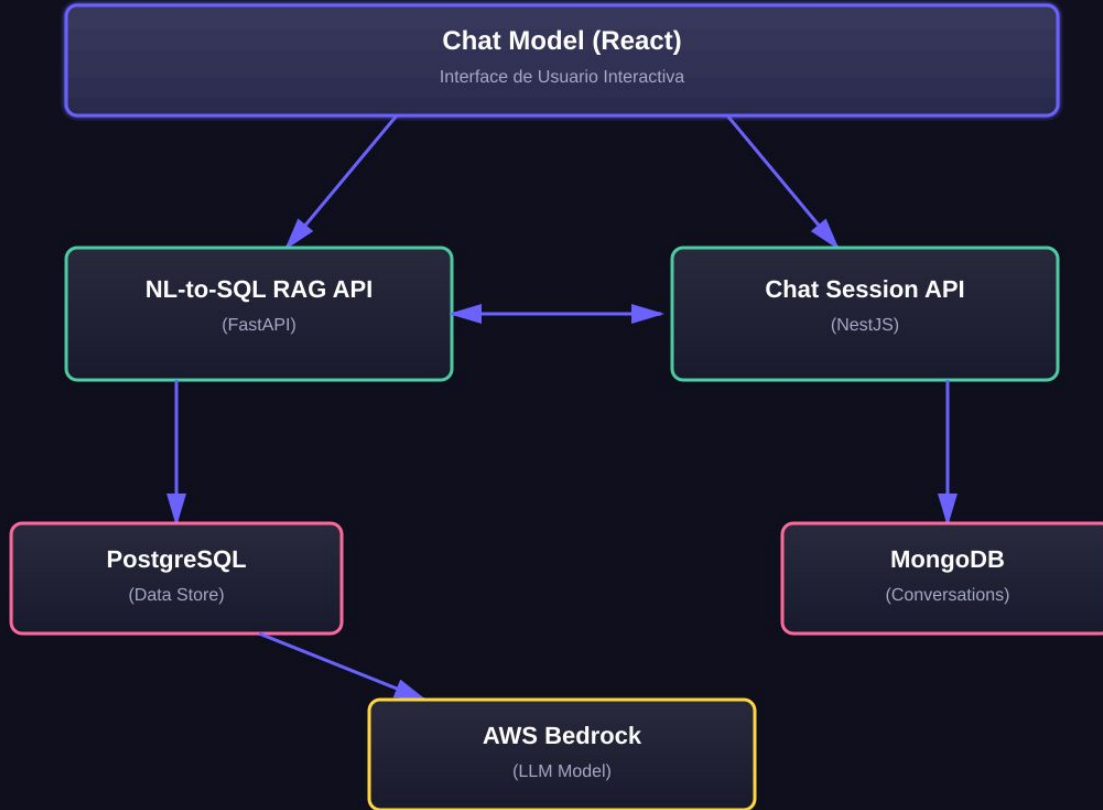
- NestJS - Framework de Node.js
- MongoDB - Base de datos de conversaciones
- Mongoose - ODM para MongoDB
- class-validator - Validación de datos

Frontend:

- React 18 - Biblioteca de UI
- TypeScript - Tipado estático
- Vite - Build tool
- Ant Design - Componentes de UI
- React Syntax Highlighter - Resaltado de código SQL

|

Arquitectura del Sistema



API / Flow

Services

Databases

AI/ML



Arquitectura de Flujo - DataQuery AI

1 - Pregunta - Usuario:

El usuario formula consultas en lenguaje cotidiano sin conocimiento técnico de SQL o estructura de base de datos.

Ejemplos:

- "Ventas por región último trimestre"
- "Top 10 productos más vendidos"
- "Comparar crecimiento mensual por categoría"

Características:

- Lenguaje natural español
- Sin sintaxis técnica requerida
- Preguntas empresariales del mundo real
- Contexto empresarial implícito



2 - Contexto - RAG + Esquema BD:

El sistema recupera y inyecta información del esquema de base de datos para contextualizar la pregunta.

Proceso:

- Analiza tablas y columnas relevantes
- Identifica relaciones entre entidades
- Recupera metadata de estructura de datos
- Enriquece el prompt con contexto específico

Beneficios:

- SQL más preciso y contextualizado
- Evita ambigüedades en nombres de columnas
- Mejora comprensión de relaciones entre datos

3 - Gráfico - Detección Automática ML:

Modelo de machine learning fine-tuned que analiza la consulta y resultados para determinar visualización óptima.

Detección Inteligente:

- ¿La pregunta implica comparación? → Gráfico de barras
- ¿Muestra tendencia temporal? → Gráfico lineal
- ¿Representa proporciones? → Gráfico circular
- ¿Analiza correlaciones? → Gráfico de dispersión



4 - SQL - AI Generativa

Modelos de lenguaje especializados convierten la pregunta enriquecida en consultas SQL ejecutables.

Capacidades:

- Genera JOINS automáticos basados en relaciones
- Aplica filtros WHERE apropiados
- Selecciona funciones de agregación (SUM, COUNT, AVG)
- Incluye ordenamiento y agrupamiento lógico
- Detecta necesidad de visualización automática

Output: SQL listo para ejecución + metadata adicional

5 - Validación - Seguridad Multi-capa:

El sistema recupera y inyecta información del esquema de base de datos para contextualizar la pregunta.

Sistema de seguridad proactivo que garantiza consultas seguras antes de la ejecución.

Capas de Validación:

- Sintáctica: Solo consultas SELECT permitidas
- Patrones Peligrosos: Bloqueo de DROP, DELETE, etc.
- Longitud Controlada: Prevención de consultas infinitas
- Límites Automáticos: LIMIT aplicado si no existe
- Sanitización: Escape de caracteres peligrosos

Resultado: SQL 100% seguro para ejecución



6 - Datos ← Ejecución BD:

Ejecución optimizada de la consulta SQL en la base de datos empresarial con manejo robusto de resultados.

Proceso:

- Conexión segura a BD
- Ejecución de consulta validada
- Transformación de resultados
- Manejo de errores y timeouts

7 - Generación de Gráficos

Generación Automática:

- Código matplotlib/seaborn dinámico
- Conversión a base64 para API
- Fallback graceful en errores

Fine-Tuning: DistilBERT (Transformers)

Objetivo:

Entrenar un modelo capaz de clasificar consultas de texto en distintas categorías utilizando técnicas modernas de NLP.

```
# Cargar y preparar los datos
df = pd.read_csv('consultas_entrenamiento_modelo_mejorado.csv')

# Ver las primeras filas
print("Primeras filas del dataset:")
print(df.head())
print(f"\nTamaño del dataset: {len(df)}")

# Explorar las clases
print("\nDistribución de necesita_grafico:")
print(df['necesita_grafico'].value_counts())
print("\nDistribución de tipo_grafico:")
print(df['tipo_grafico'].value_counts())
```

```
tipo_grafico
0  histograma
1   barras
2   barras
3   barras
4  histograma

Tamaño del dataset: 2500
```

```
Distribución de tipo_grafico:
tipo_grafico
barras      1362
lineal      372
circular    258
histograma  182
ninguno     178
dispersion  148
Name: count, dtype: int64
```

Flujo de trabajo:

Instalación de dependencias

- Transformers, Torch, Scikit-learn, Pandas, NumPy.

Carga y exploración del dataset

- Archivo:
consultas_entrenamiento_modelo_mejorado.csv
- Análisis de tamaño y distribución de clases.

```
Distribución de necesita_grafico:
necesita_grafico
True      1177
1         1145
False     139
0          39
Name: count, dtype: int64
```

```
# Preprocesamiento de datos
def preprocess_text(text):
    """Limpia y preprocesa el texto de las consultas"""
    if isinstance(text, str):
        # Convertir a minúsculas
        text = text.lower()
        # Remover caracteres especiales pero mantener espacios y letras
        text = re.sub(r'[^w\s]', '', text)
        return text.strip()
    return ""

# Aplicar preprocesamiento
df['consulta_limpia'] = df['consulta'].apply(preprocess_text)
```

Preprocesamiento del texto

- Limpieza con expresiones regulares.
- Se procesa y normaliza etiquetas para determinar qué tipo de gráfico debe generarse.
- División en train/validation.

```
# Procesamiento de las etiquetas
def create_improved_label(row):
    necesita_grafico = row['necesita_grafico']
    tipo_grafico = row['tipo_grafico']

    # Convertir a valores booleanos consistentes
    if necesita_grafico in [True, 'True', '1', 1, 'true']:
        necesita_grafico_bool = True
    else:
        necesita_grafico_bool = False

    # Si no necesita gráfico, siempre es 'ninguno'
    if not necesita_grafico_bool:
        return 'ninguno'

    # Si necesita gráfico pero el tipo es 'ninguno', revisar el contexto
    if tipo_grafico == 'ninguno' and necesita_grafico_bool:
        # Revisar si la consulta realmente pide un gráfico
        consulta = str(row['consulta']).lower()
```

```
# =====
# 5 División train / val
# =====
train_texts, val_texts, train_labels, val_labels = train_test_split(
    df['consulta_limpia'].values,
    df['label'].values,
    test_size=0.2,
    random_state=42,
    stratify=df['label'].values
)

print(f"\nTamaño del conjunto de entrenamiento: {len(train_texts)}")
print(f"Tamaño del conjunto de validación: {len(val_texts)}")
```

Tamaño del conjunto de entrenamiento: 2000
Tamaño del conjunto de validación: 500

```
# =====
# CONFIGURACIÓN DE MODELOS
# =====

# Modelo 1: DistilBERT (original)
model_name_distilbert = 'distilbert-base-uncased'
tokenizer_distilbert = DistilBertTokenizer.from_pretrained(model_name_distilbert)
num_classes = len(label_mapping)

model_distilbert = DistilBertForSequenceClassification.from_pretrained(
    model_name_distilbert,
    num_labels=num_classes
)
```

```
# Modelo 2: bert-base-spanish-wwm-cased
model_name_beto = "dccuchile/bert-base-spanish-wwm-cased"
tokenizer_beto = AutoTokenizer.from_pretrained(model_name_beto)

model_beto = AutoModelForSequenceClassification.from_pretrained(
    model_name_beto,
    num_labels=num_classes
)
```

Se configura el modelo DistilBERT para clasificación de texto. Primero define el nombre del modelo base como 'distilbert-base-uncased', que es una versión compacta y eficiente del modelo BERT original que procesa el texto en minúsculas. Luego inicializa el tokenizador específico para DistilBERT, que se encarga de convertir el texto en tokens comprensibles para el modelo.

Posteriormente, determina el número de clases de clasificación basándose en la longitud del mapeo de etiquetas definido previamente. Finalmente, crea el modelo de clasificación cargando la versión pre-entrenada de DistilBERT y adaptándola para la tarea específica de clasificación de secuencias, configurando la capa final con el número exacto de categorías que necesita predecir.

La configuración prepara el modelo para ser fine-tuned en tareas de clasificación de texto como análisis de sentimientos o categorización de documentos, partiendo de los pesos pre-entrenados y añadiendo una capa de clasificación personalizada para el problema específico.

```
def ejecutar_validacion_cruzada(model_class, tokenizer, model_name, texts, labels, device,
                               n_splits=5, epochs=3, batch_size=16):
    """
    Ejecuta validación cruzada k-fold para un modelo específico
    """
    print(f"\n{'='*60}")
    print(f"VALIDACIÓN CRUZADA {n_splits}-FOLD: {model_name}")
    print(f"{'='*60}")

    # Configurar k-fold
    kf = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=42)

    # Almacenar resultados
    fold_results = {
        'accuracies': [],
        'losses': [],
        'predictions': [],
        'actual_labels': [],
        'fold_models': []
    }

    fold_num = 1

    for train_idx, val_idx in kf.split(texts, labels):
        print(f"\n🌀 Fold {fold_num}/{n_splits}")

        # Dividir datos
        train_texts_fold = texts[train_idx]
        train_labels_fold = labels[train_idx]
        val_texts_fold = texts[val_idx]
        val_labels_fold = labels[val_idx]
```

Proceso Principal:

1 - Configuración Inicial

- Crea divisores k-fold estratificados para preservar distribución de clases
- Prepara estructura para almacenar resultados de cada fold

2 - Procesamiento por Fold

- Divide datos en entrenamiento/validación para cada partición
- Crea datasets y data loaders para manejo eficiente de lotes
- Inicializa modelo nuevo para cada fold

3 - Entrenamiento por Épocas


- Configura optimizador AdamW y programador de learning rate
- Ejecuta ciclos de entrenamiento y evaluación por épocas
- Realiza seguimiento del mejor accuracy por fold

4 - Recolección de Resultados

- Almacena métricas de precisión y pérdida
- Guarda predicciones y etiquetas reales
- Conserva pesos entrenados de cada modelo fold

Características Clave:

- Evaluación robusta con múltiples particiones de datos
- Entrenamiento independiente por fold
- Métricas comparativas para evaluación de modelo
- Configuración reproducible con semilla aleatoria fija

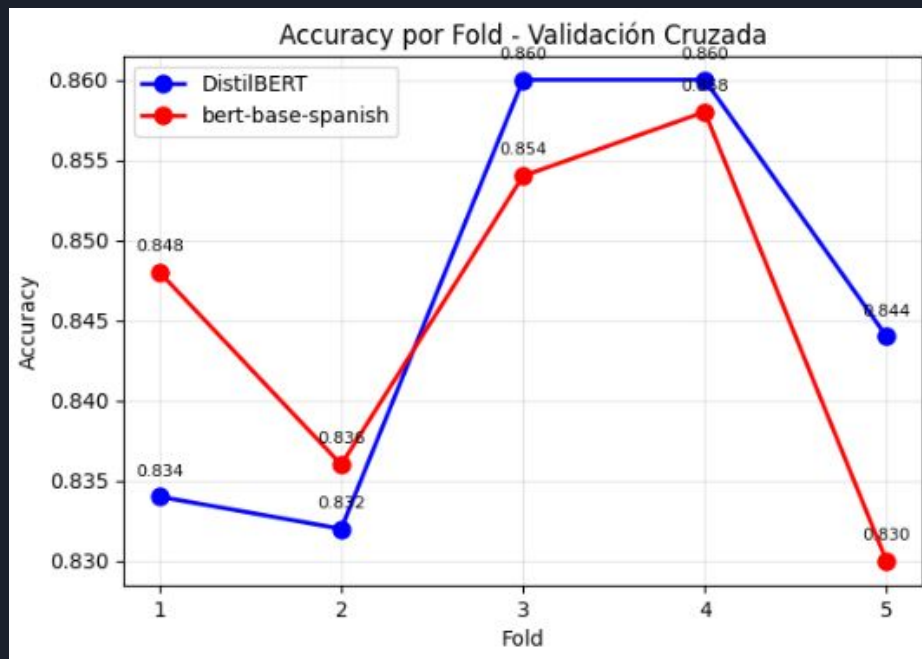


```
# =====
# EJECUTAR VALIDACIÓN CRUZADA PARA AMBOS MODELOS
# =====

print(f"\n{'='*60}")
print("INICIANDO VALIDACIÓN CRUZADA COMPARATIVA")
print(f"{'='*60}")

# Ejecutar validación cruzada para DistilBERT
cv_results_distilbert = ejecutar_validacion_cruzada(
    DistilBertForSequenceClassification,
    tokenizer_distilbert,
    'distilbert-base-uncased',
    df['consulta_limpia'].values,
    df['label'].values,
    device,
    n_splits=5,
    epochs=3,
    batch_size=16
)

# Ejecutar validación cruzada para bert-base-spanish-wwm-cased
cv_results_beto = ejecutar_validacion_cruzada(
    AutoModelForSequenceClassification,
    tokenizer_beto,
    'dccuchile/bert-base-spanish-wwm-cased',
    df['consulta_limpia'].values,
    df['label'].values,
    device,
    n_splits=5,
    epochs=3,
    batch_size=16
)
```



El gráfico muestra la evolución del accuracy en una validación cruzada para los modelos DistilBERT y bert-base-spanish. Ambos alcanzan valores altos, en torno al 0.84 y 0.86, lo que indica un buen desempeño general. Sin embargo, se observan diferencias en la estabilidad y consistencia de los resultados.

El modelo bert-base-spanish obtiene una leve ventaja en los dos primeros folds, pero en los tres restantes DistilBERT logra mejores resultados y con una mayor regularidad. Mientras que DistilBERT mantiene su precisión dentro de un rango acotado entre 0.832 y 0.860, el modelo bert-base-spanish muestra más variabilidad, descendiendo hasta 0.830 en el último fold.

Resumen Comparativo de Métricas

Métrica	DistilBERT	Beto	Diferencia
Accuracy	0.8460	0.8452	-0.0008
Std Dev	0.0121	0.0106	-0.0015
Coef. Var.	1.43%	1.26%	-0.18%
Macro F1	0.7884	0.7928	+0.0044
Weighted F1	0.8396	0.8372	-0.0024

La tabla presenta una comparación de métricas de rendimiento entre los modelos DistilBERT y Beto. Ambos muestran resultados muy similares, lo que sugiere un desempeño equivalente en términos generales. El accuracy de DistilBERT es apenas superior, aunque la diferencia es mínima, prácticamente despreciable desde el punto de vista estadístico. Sin embargo, su desviación estándar es ligeramente mayor, lo que indica una leve menor consistencia en los resultados. El coeficiente de variación refuerza esta observación, ya que DistilBERT muestra un porcentaje un poco más alto, señalando una variabilidad relativa mayor. En cuanto a las métricas de F1, el modelo DistilBERT obtiene un mejor puntaje en el Macro F1, lo que sugiere un desempeño más equilibrado entre las distintas clases, mientras que Beto logra un resultado marginalmente superior en el Weighted F1, favorecido posiblemente por un mejor rendimiento en las clases más frecuentes.

En conjunto, ambos modelos tienen un rendimiento muy competitivo, con ventajas mínimas en distintos aspectos. DistilBERT destaca por un mejor balance entre clases y Beto por una ligera estabilidad y precisión ponderada.

Sistema basado en reglas para selección de esquemas de base de datos

```
class SchemaSelector:
    """
    Selección de esquema basada en reglas + keywords.
    Mucho más rápido y predecible que embeddings semánticos.
    """

    def __init__(self, db_service):
        self.db_service = db_service
        self.tables_metadata: Dict[str, TableMetadata] = {}
        self.keyword_index: Dict[str, Set[str]] = {}
        self._build_indexes()
```

```
def _extract_schema_from_db(self) -> Dict:
    """Extrae el esquema completo de la base de datos"""
    try:
        # Obtener todas las tablas
        tables_query = """
            SELECT table_name
            FROM information_schema.tables
            WHERE table_schema = 'public'
            AND table_type = 'BASE TABLE'
            ORDER BY table_name
        """
        tables_result = self.db_service.execute_query(tables_query)
        tables = [row[0] for row in tables_result["data"]]

        schema_info = {}

        for table in tables:
            # Obtener columnas y tipos
            columns_query = f"""
                SELECT column_name, data_type, is_nullable
                FROM information_schema.columns
                WHERE table_name = '{table}'
                ORDER BY ordinal_position
            """
            columns_result = self.db_service.execute_query(columns_query)

            # Obtener relaciones (FKs)
            relations_query = f"""
```

Es la clase principal del sistema.

Contiene la lógica para:

- Extraer el esquema de la base de datos,
- Construir índices de búsqueda,
- Puntuar y seleccionar tablas relevantes,
- Generar un contexto textual con la información de esas tablas.

Extrae toda la estructura de la base de datos desde information_schema:

obtiene las tablas, columnas y tipos, las relaciones (foreign keys), y los comentarios descriptivos de tablas y columnas.

Devuelve un diccionario con toda esta información, que luego servirá para generar metadatos y contexto.

```

def _extract_keywords(self, table_name: str, table_info: Dict) -> Set[str]:
    """Extrae keywords relevantes de una tabla"""
    keywords = set()

    # Nombre de tabla (singular/plural)
    keywords.add(table_name.lower())
    keywords.add(table_name.rstrip("s").lower())
    keywords.update(table_name.lower().split("_"))

    # Keywords de descripción
    if table_info.get("table_comment"):
        desc_words = re.findall(r"\b\w{4,}\b", table_info["table_comment"].lower())
        keywords.update(desc_words[:10])

    # Nombres de columnas importantes
    skip_columns = {"id", "created_at", "updated_at", "activo"}
    for col in table_info.get("columns", []):
        col_name = col["name"].lower()
        if col_name not in skip_columns:
            keywords.add(col_name)
            keywords.update(col_name.split("_"))

    # Keywords de comentarios de columnas
    for col in table_info.get("columns", []):
        if col.get("comment"):
            comment_words = re.findall(r"\b\w{4,}\b", col["comment"].lower())
            keywords.update(comment_words[:5])

    return keywords

```

Extrae las palabras clave que describen mejor cada tabla.

Proviene de:

- el nombre de la tabla (en singular/plural),
- las columnas,
- los comentarios descriptivos,
- las palabras significativas (sin stopwords).

Permite que una búsqueda como "clientes activos" matchee con la tabla clientes aunque el usuario no use exactamente el nombre de la tabla.

```

def _build_indexes(self):
    """Construye índices de búsqueda rápida"""
    schema = self._extract_schema_from_db()

    for table_name, table_info in schema.items():
        keywords = self._extract_keywords(table_name, table_info)

        metadata = TableMetadata(
            name=table_name,
            keywords=keywords,
            columns=[col["name"] for col in table_info["columns"]],
            relationships=self._extract_relationships(table_info),
            description=table_info.get("table_comment", ""),
        )

        self.tables_metadata[table_name] = metadata

    # Indexar keywords -> tablas
    for keyword in keywords:
        if keyword not in self.keyword_index:
            self.keyword_index[keyword] = set()
        self.keyword_index[keyword].add(table_name)

    logger.info(
        f"✅ Índices contruidos: {len(self.tables_metadata)} tablas, "
        f"{len(self.keyword_index)} keywords"
    )

```

Crea índices que permiten buscar rápido las tablas relevantes:

- 1 - Llama a `_extract_schema_from_db()`.
- 2 - Para cada tabla, genera sus keywords con `_extract_keywords()`.
- 3 - Crea un objeto `TableMetadata`.
- 4 - Llena dos estructuras:
 - `tables_metadata`: info detallada por tabla.
 - `keyword_index`: mapa de keyword → conjunto de tablas que la contienen.

```

def _score_table(self, table_name: str, query_tokens: List[str]) -> float:
    """Calcula score de relevancia para una tabla"""
    if table_name not in self.tables_metadata:
        return 0.0

    metadata = self.tables_metadata[table_name]
    score = 0.0

    for token in query_tokens:
        # Match exacto con nombre de tabla
        if token in table_name.lower():
            score += 10.0

        # Match con keywords
        if token in metadata.keywords:
            score += 5.0

        # Match con columnas
        if token in [col.lower() for col in metadata.columns]:
            score += 3.0

        # Match en descripción
        if metadata.description and token in metadata.description.lower():
            score += 1.0

    return score

```

Asigna un puntaje de relevancia a cada tabla en función de la query:

+10 si el token aparece en el nombre de la tabla,

+5 si aparece entre las keywords,

+3 si coincide con una columna,

+1 si aparece en la descripción.

Cuanto más fuerte sea el match, mayor la relevancia.

```

def select_relevant_tables(
    self, query: str, max_tables: int = 5
) -> List[Tuple[str, float]]:
    """Selecciona tablas relevantes para la query"""
    query_tokens = self._tokenize_query(query)

    # Búsqueda por keywords
    candidate_tables = set()
    for token in query_tokens:
        if token in self.keyword_index:
            candidate_tables.update(self.keyword_index[token])

    # Si no hay candidatos, usar todas
    if not candidate_tables:
        candidate_tables = set(self.tables_metadata.keys())

    # Calcular scores
    table_scores = []
    for table_name in candidate_tables:
        score = self._score_table(table_name, query_tokens)
        if score > 0:
            table_scores.append((table_name, score))

    table_scores.sort(key=lambda x: x[1], reverse=True)

    # Expandir con tablas relacionadas
    final_tables = set()
    for table_name, score in table_scores[:max_tables]:
        final_tables.add(table_name)

        metadata = self.tables_metadata[table_name]
        for related_table in metadata.relationships.values():
            if len(final_tables) < max_tables * 2:
                final_tables.add(related_table)

    # Recalcular scores
    final_scores = []
    for table_name in final_tables:
        score = self._score_table(table_name, query_tokens)
        final_scores.append((table_name, score))

```

Es el núcleo del sistema de selección:

- Tokeniza la consulta.
- Busca tablas candidatas según keywords.
- Calcula los puntajes de relevancia.
- Elige las más relevantes (hasta max_tables).
- Expande el conjunto con las tablas relacionadas por FK.
- Devuelve una lista ordenada de (tabla, score).


```

enhanced_prompt = f"""
# CONTEXTO DE BASE DE DATOS (ESQUEMA ESTRICTO)
{schema_context}

# REQUISITOS DE VISUALIZACIÓN
- Necesita gráfico: {chart_requirements['needs_chart']}
- Tipo de gráfico sugerido: {chart_requirements['chart_type']}
- Confianza: {chart_requirements['confidence']}
- Razón: {chart_requirements['reasoning']}

# INSTRUCCIONES GENERALES
Eres un experto en PostgreSQL y visualización de datos. Tu tarea es gene

# REGLAS CRÍTICAS (OBLIGATORIAS)
1. **Usa SOLO** las tablas y columnas mencionadas en el esquema del cont
    - Si el esquema no incluye una columna o tabla, **no la inventes** ni
    - Si no hay suficiente información para responder, **devuelve un erro
2. Usa los nombres exactos de tablas y columnas tal como aparecen en el.
3. Si la consulta requiere un JOIN, solo usa relaciones foreign key infe
4. Optimiza para rendimiento, pero prioriza exactitud sobre optimizaciór
5. Considera los requisitos de visualización (tipo de gráfico, ejes, cat
6. No uses alias o funciones sobre columnas inexistentes.
7. No inventes métricas o agregaciones no justificadas.

# MANEJO DE ERRORES
Si la consulta no puede generarse sin violar las reglas anteriores, resp

{{
    "error": "No se puede generar la consulta sin inventar columnas o ta
    "reason": "Explicación breve del motivo."
}}

# FORMATO DE RESPUESTA (CRÍTICO)
**Responde EXCLUSIVAMENTE con JSON válido (sin markdown, sin ``json, si

{{
    "sql_query": "Consulta SQL aquí",
    "needs_chart": {str(chart_requirements['needs_chart']).lower()},
    "chart_type": "bar|line|scatter|pie|histogram|area|box_plot|null",

```

Propósito Principal

Crear un prompt que guíe a un modelo de IA para:

Generar consultas SQL válidas para PostgreSQL

Crear código de visualización en Python (matplotlib/seaborn)

Validar que solo use el esquema de base de datos proporcionado

Componentes Clave

1. Contexto de Base de Datos

inyecta el esquema de tablas y columnas disponibles

2. Requisitos de Visualización

Define si necesita gráfico, tipo sugerido, confianza y razonamiento

3. Reglas Estrictas

- No inventar tablas o columnas no existentes
- Usar nombres exactos del esquema
- Priorizar exactitud sobre optimización

4. Manejo de Errores

Formato JSON estandarizado para errores cuando no se puede generar la consulta

5. Formato de Respuesta Estructurado

6. Especificaciones de Gráficos

Mapea tipos de gráfico a sus requisitos de datos

Proporciona ejemplos de código Python para visualizaciones

Estructura del proyecto



Backend

API Principal

api_model_fast/

app/

main.py

routes/

rag_api.py

health.py

services/

rag/

rag_pipeline.py

schema_selector.py

bedrock_service.py

chart_detector.py

chart_service.py

security_validator.py

models/

modelo_distilbert_mejorado/

config/

config.py

requirements.txt



Frontend

Interfaz de Usuario

chat_model/

src/

components/

Chat/

Charts/

Sidebar/

UI/

hooks/

services/

types/

utils/

package.json

vite.config.ts



Persistencia

Gestión de Sesiones

chat_session/

src/

main.ts

app.module.ts

conversations/

conversations.controller.ts

conversations.service.ts

dto/

schemas/

interfaces/

package.json

Sistema de Gestión de Ventas

Base de Datos PostgreSQL



CLIENTES

- dni_ruc
- nombres
- apellidos
- email
- teléfono



PRODUCTOS

- sku
- nombre
- precio_venta
- stock_actual
- categoría



COMPROBANTES_VENTA

- número
- fecha
- subtotal
- igv
- total



TRANSACCIONES

- método_pago
- tarjeta
- monto
- estado



DESCUENTOS

- tipo
- valor
- fecha_inicio
- fecha_fin



PROVEEDORES

- ruc
- razón_social
- contacto

Características Principales



Gestión de Ventas

Comprobantes, detalles y transacciones



Control de Inventario

Stock, movimientos y auditoría



Múltiples Pagos

Efectivo, tarjetas y planes de cuotas



Promociones

Descuentos por porcentaje o monto fijo