Facsímil 04

# La caja de herramientas

Capítulo 12

## Text-to-SQL y herramientas de datos

686f6c61 · 2026

# Capítulo 12: Text-to-SQL y herramientas de datos

## La pregunta que no está en un documento

En el capítulo 11 vimos que un sistema puede decidir consultar distintas fuentes antes de responder. Algunas fuentes son textos: normativas, manuales, tickets, actas. Otras no son textos en sentido estricto: tablas, bases de datos, métricas, historiales, estados de pedidos, matrículas, pagos, sensores o registros de producto.

Ahí aparece una frontera importante. Un RAG puede explicar "qué dice la política de matrícula". Pero si preguntas "¿cuántos alumnos tienen pago pendiente y beca en revisión?", no quieres que el modelo imagine un número a partir de párrafos. Quieres que consulte datos.

Text-to-SQL nace de esa necesidad: convertir una pregunta humana en una consulta SQL controlada. No es "hablar con la base de datos como si fuera una persona". Es construir un puente auditado entre intención, esquema, permisos, consulta, ejecución, resultado y explicación.

## Estado del arte con fecha de corte

**Fecha de corte:** 26 de mayo de 2026.
**Fuentes consultadas ese día:** papers de Spider, RAT-SQL, PICARD, BIRD y Spider 2.0; documentación oficial de LangChain SQL Agent, LlamaIndex NL SQL, OpenAI Function Calling y Structured Outputs; documentación de SQLGlot, DuckDB y repositorio público de Vanna.

Text-to-SQL se estudia desde hace años como una tarea de *semantic parsing*: traducir lenguaje natural a una representación formal ejecutable. Spider marcó un salto porque propuso preguntas y consultas complejas sobre 200 bases de datos, con esquemas distintos entre entrenamiento y prueba; su objetivo era medir generalización a bases nuevas, no memorizar una sola tabla.[1]

Después se vio que una parte crítica no era solo generar SQL, sino entender el esquema. RAT-SQL trabajó explícitamente el *schema linking*: conectar palabras de la pregunta con tablas, columnas y relaciones del esquema mediante atención consciente de relaciones.[2] PICARD atacó otro problema: incluso un modelo capaz puede producir SQL inválido. Su propuesta restringe la decodificación con parsing incremental para rechazar tokens que romperían la sintaxis formal.[3]

Los benchmarks más recientes empujaron la tarea hacia condiciones más reales. BIRD introdujo bases de datos grandes, valores sucios, conocimiento externo y eficiencia de consulta; reportó 12.751 pares pregunta-SQL sobre 95 bases de datos y 33,4 GB.[4] Spider 2.0 fue más allá: problemas de flujo empresarial, más de 1.000 columnas en algunas bases, varios dialectos como BigQuery y Snowflake, metadatos extensos y tareas que pueden requerir múltiples consultas.[5]

En herramientas prácticas, LangChain documenta un flujo de SQL agent que lista tablas, decide cuáles son relevantes, inspecciona esquemas, genera consulta, revisa errores comunes, ejecuta y formula respuesta.[6] LlamaIndex ofrece componentes NL SQL y advierte que ejecutar consultas generadas requiere especial cuidado con permisos y entorno.[7] OpenAI documenta function calling como forma de describir herramientas mediante esquemas y recibir argumentos estructurados; Structured Outputs permite exigir que una salida siga un esquema JSON compatible.[8]

---

## Qué no es Text-to-SQL

Text-to-SQL no es “darle acceso libre a la base de datos al modelo”. El modelo no debería decidir por su cuenta qué puede leer, cuántas filas puede sacar, qué tablas puede cruzar o si una consulta es aceptable. Eso lo decide el sistema.

Tampoco es un reemplazo completo de un equipo de datos. Muchas preguntas parecen simples y esconden definiciones de negocio: “cliente activo”, “ingreso neto”, “alumno matriculado”, “churn”, “pedido completado”. Si esas definiciones no viven en una capa semántica o en documentación recuperable, el modelo puede generar una consulta válida que responde a otra pregunta.

Y no es solo una tarea de SQL. En producción intervienen permisos, catálogo, metadatos, documentación de columnas, dialecto, coste, timeouts, límites de filas, validación, trazas y evaluación. La consulta es una pieza. El sistema es todo lo que evita que esa pieza se use mal.

---

## Qué sí es

Text-to-SQL es una tubería controlada:

```
pregunta -> intención -> esquema relevante -> SQL candidato
         -> validación -> ejecución limitada -> resultado
         -> explicación con trazas
```

La unidad de trabajo no es “una query”. La unidad de trabajo es una **solicitud de datos** con contexto:

| PIEZA | QUÉ CONTIENE | EJEMPLO |
|---|---|---|
| Pregunta | Lo que pide la persona. | “Ingresos por campus en marzo”. |
| Usuario | Identidad, rol y permisos. | `analista_matricula`, campus permitido. |
| Dominio | Área de datos. | Matrícula, becas, pagos, soporte. |
| Esquema | Tablas, columnas, claves y tipos. | `pagos`, `alumnos`, `campus`. |
| Semántica | Definiciones de negocio. | “ingreso neto = importe - devoluciones”. |
| SQL | Consulta candidata. | `SELECT campus, SUM(...) ...`. |
| Validación | Reglas antes de ejecutar. | Solo `SELECT`, `LIMIT`, timeout. |
| Resultado | Filas devueltas. | Tabla agregada. |
| Explicación | Resumen humano. | “Campus Norte concentra el 42%”. |
| Traza | Registro completo. | Tablas usadas, SQL, coste, tiempo. |

Para entenderlo bien, pensemos en una pregunta sencilla:

> “¿Cuáles son los tres campus con más pagos pendientes?”

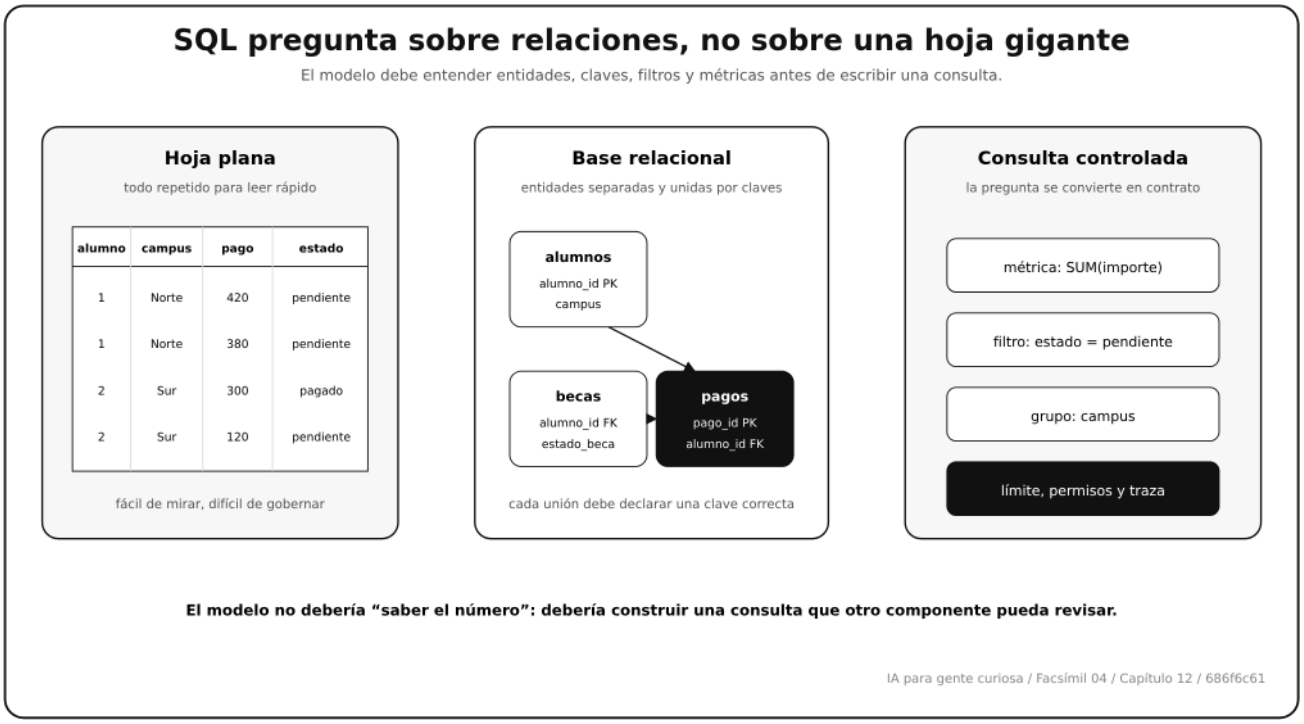
El sistema no debería saltar directamente a escribir SQL. Primero debe saber qué significa “pagos pendientes”, dónde vive “campus”, si la persona puede ver todos los campus, si hay que excluir pagos anulados, si la moneda importa y si “tres” implica ordenar de mayor a menor.

---

## SQL desde cero, pero sin rebajarlo

SQL es el lenguaje clásico para consultar datos relacionales. Una base relacional organiza información en tablas. Cada tabla tiene filas y columnas. Una fila representa una entidad o evento; una columna representa un atributo.

Una base de datos no es un Excel grande. En una hoja plana es habitual repetir datos para que todo quepa en una misma vista. En una base relacional se separan entidades y eventos para que cada cosa tenga un lugar claro: alumnos por un lado, pagos por otro, becas por otro. Después una consulta une lo que necesita mediante claves.




Ejemplo mínimo:

| TABLA `PAGOS` | SIGNIFICADO |
|---|---|
| `pago_id` | identificador del pago. |
| `alumno_id` | alumno asociado. |
| `campus` | campus administrativo. |
| `estado` | `pagado`, `pendiente`, `devuelto`. |
| `importe` | cantidad. |
| `fecha` | fecha del movimiento. |

Una consulta básica:

```
SELECT campus, SUM(importe) AS total_pendiente
FROM pagos
WHERE estado = 'pendiente'
GROUP BY campus
ORDER BY total_pendiente DESC
LIMIT 3;
```

Esto se lee así:

| CLÁUSULA | QUÉ HACE | EN CASTELLANO |
|---|---|---|
| `SELECT` | Elige columnas o cálculos. | Quiero campus y suma de importe. |
| `FROM` | Indica la tabla base. | Usa la tabla de pagos. |
| `WHERE` | Filtra filas antes de agrupar. | Solo pagos pendientes. |
| `GROUP BY` | Agrupa filas por una dimensión. | Junta pagos por campus. |
| `ORDER BY` | Ordena resultados. | Primero los importes mayores. |
| `LIMIT` | Limita filas devueltas. | Dame solo tres. |

La parte que más se suele subestimar es el `JOIN`: cruzar tablas. Si `pagos` tiene `alumno_id`, pero la titulación vive en `alumnos`, necesitamos unir:

```
SELECT a.titulacion, SUM(p.importe) AS total_pendiente
FROM pagos AS p
JOIN alumnos AS a ON a.alumno_id = p.alumno_id
WHERE p.estado = 'pendiente'
GROUP BY a.titulacion
ORDER BY total_pendiente DESC
LIMIT 5;
```

Un `JOIN` no es decoración. Es una afirmación sobre relación entre tablas. Si la clave es incorrecta, el resultado puede ser válido en SQL y falso en negocio.

## Errores SQL que cambian la historia

Para un ingeniero, el peligro no está solo en que el modelo genere SQL inválido. Ese fallo es fácil de detectar. El peligro serio es que genere SQL válido, rápido y aparentemente razonable, pero calcule otra cosa.

| ERROR | SQL QUE SUELE APARECER | QUÉ ROMPE | CÓMO PENSARLO |
|---|---|---|---|
| Duplicar filas con un `JOIN` | Unir `alumnos` con varias filas de `pagos`. | `COUNT(*)` sube porque un alumno aparece varias veces. | Cuenta entidades con `COUNT(DISTINCT alumno_id)`. |
| Confundir evento y entidad | Contar pagos como si fueran alumnos. | Mide movimientos, no personas. | Pregunta si cada fila es “cosa” o “suceso”. |
| Ignorar `NULL` | `AVG(importe)` sin revisar ausentes. | Algunos cálculos excluyen nulos sin avisar. | Decide si nulo significa desconocido, cero o no aplica. |
| Filtrar tarde | Usar `HAVING` cuando tocaba `WHERE`. | Agrupa datos que no deberían entrar. | `WHERE` filtra filas; `HAVING` filtra grupos. |
| Fechas mal cerradas | `fecha <= '2026-03-31'`. | Puede perder horas del último día. | Usa rangos semiabiertos: `>= inicio` y `< fin`. |
| Moneda mezclada | `SUM(importe)` sin moneda. | Suma euros, dólares o créditos como si fueran iguales. | Agrupa o convierte antes de sumar. |
| Estado de negocio incompleto | `estado != 'pagado'`. | Incluye anulados, devueltos o pruebas. | Define catálogo permitido, no solo excluido. |

Miremos el fallo de duplicación, que aparece mucho en sistemas Text-to-SQL. Si una tabla `alumnos` tiene una fila por alumno y `pagos` tiene varias filas por alumno, este SQL cuenta pagos, no alumnos:

```
SELECT COUNT(*) AS alumnos_con_pago
FROM alumnos AS a
JOIN pagos AS p ON p.alumno_id = a.alumno_id
WHERE p.estado = 'pendiente';
```

Si un alumno tiene tres pagos pendientes, aparece tres veces. La consulta puede ejecutarse sin quejarse y devolver un número bonito, pero el significado es otro. Para contar alumnos únicos:

```
SELECT COUNT(DISTINCT a.alumno_id) AS alumnos_con_pago_pendiente
FROM alumnos AS a
JOIN pagos AS p ON p.alumno_id = a.alumno_id
WHERE p.estado = 'pendiente';
```

La diferencia entre `COUNT(*)` y `COUNT(DISTINCT ...)` no es un detalle académico. Es la diferencia entre contar filas y contar entidades. Cuando una persona pregunta “cuántos alumnos”, normalmente quiere entidades. Cuando pregunta “cuántos pagos”, quiere eventos.

También importan las fechas. Si `fecha` incluye hora, esta condición parece correcta pero puede dejar fuera movimientos del 31 de marzo por la tarde:

```
WHERE fecha >= '2026-03-01'
  AND fecha <= '2026-03-31'
```

El patrón más robusto para intervalos suele ser:

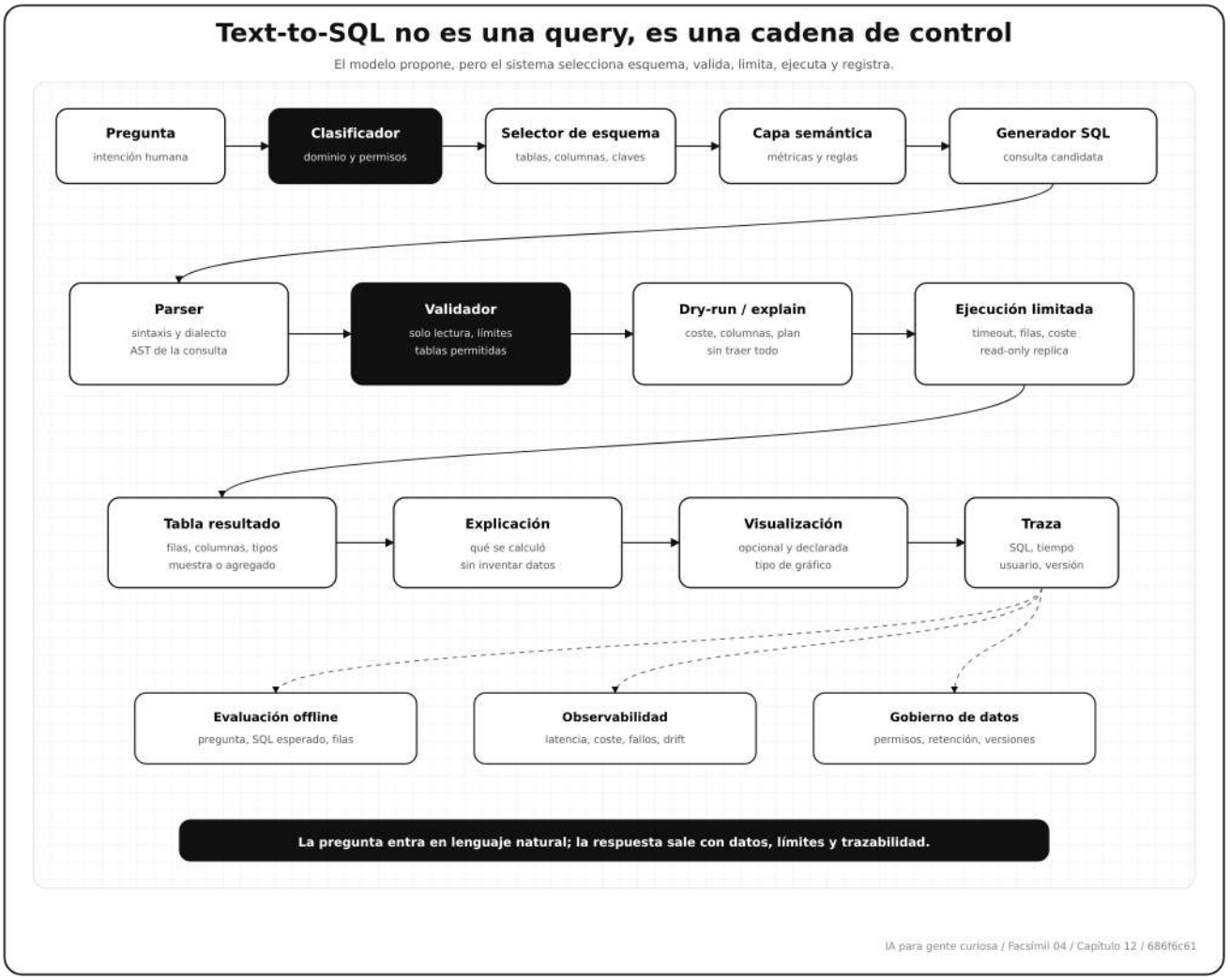
```
WHERE fecha >= '2026-03-01'
  AND fecha < '2026-04-01'
```

Un buen sistema Text-to-SQL no solo valida sintaxis. También mira estas trampas: cardinalidad de las tablas, tipo de métrica, nulos, moneda, rango temporal, catálogo de estados y claves de unión.

---

## El mecanismo paso a paso

Text-to-SQL funciona bien cuando se separan tareas. Un modelo que recibe “todas las tablas, toda la documentación y genera SQL” puede acertar en una demo pequeña, pero se vuelve frágil con esquemas grandes.




Un sistema así no depende de una única llamada al modelo. Depende de contratos entre piezas. La persona pregunta; el clasificador decide si la pregunta es de datos o de documentos; el selector reduce el esquema; la capa semántica aporta definiciones; el generador propone SQL; el validador inspecciona; la base ejecuta con límites; la respuesta explica y registra.

## Corregir el error en vez de fallar: el bucle de autocorrección

Aunque el modelo sea bueno, fallará: una columna mal escrita, un `JOIN` que no existe, una función de un dialecto equivocado. La diferencia entre un prototipo y un sistema serio es qué pasa entonces. La pieza que lo distingue es el **bucle de autocorrección**: cuando el validador rechaza el SQL o la base de datos devuelve un error, ese error se le **devuelve al modelo** junto con el SQL fallido para que lo arregle, en lugar de propagar el fallo al usuario.

El mensaje de error de una base de datos es muy informativo (`no such column: paid_at`, `ambiguous column name: campus`), y el modelo suele corregir a la primera o segunda con esa pista. Pero hay dos reglas de ingeniería innegociables:

- **Tope de reintentos.** Sin un máximo (dos o tres), un SQL que el modelo no sabe arreglar genera un bucle de llamadas y factura sin fin. Igual que el agente del capítulo 11: sin límite, no hay sistema.
- **Solo se reintenta lo seguro.** Se reintenta sobre errores de *sintaxis o esquema*, validando de nuevo cada candidato antes de ejecutar. Nunca se reintenta «a ciegas» ejecutando una y otra vez contra la base de datos de producción.

Cuando los reintentos se agotan, la respuesta correcta no es inventar: es **abstenerse** y, si procede, pedir aclaración. Un Text-to-SQL que se rinde con un «no he podido construir una consulta fiable para esto» es mejor que uno que devuelve un número que nadie puede defender.

---

## El problema real: esquema, semántica y valores

Cuando un humano experto escribe SQL, no solo recuerda sintaxis. Recuerda el significado de cada tabla, qué columnas son fiables, qué claves se unen, qué estados se excluyen, qué fechas mandan y qué métricas no se calculan directamente.

Text-to-SQL falla por tres razones principales:

| FOCO | QUÉ FALLA | EJEMPLO |
|---|---|---|
| Esquema | El modelo elige tabla o columna incorrecta. | Usa `created_at` en vez de `paid_at`. |
| Semántica | La consulta no respeta definición de negocio. | Cuenta alumnos anulados como activos. |
| Valores | No sabe cómo están escritos los datos reales. | Busca `pendiente` pero la tabla usa `PENDING`. |

El *schema linking* conecta pregunta y esquema. Si la pregunta dice “campus con pagos pendientes”, el sistema debe vincular:

| PALABRA DE LA PREGUNTA | CANDIDATO EN DATOS | POR QUÉ |
|---|---|---|
| campus | `alumnos.campus` o `pagos.campus` | Dimensión de agrupación. |
| pagos | tabla `pagos` | Hecho económico. |
| pendientes | `pagos.estado = 'pendiente'` | Filtro. |
| tres | `LIMIT 3` | Tamaño de salida. |
| más | `ORDER BY total DESC` | Orden descendente. |

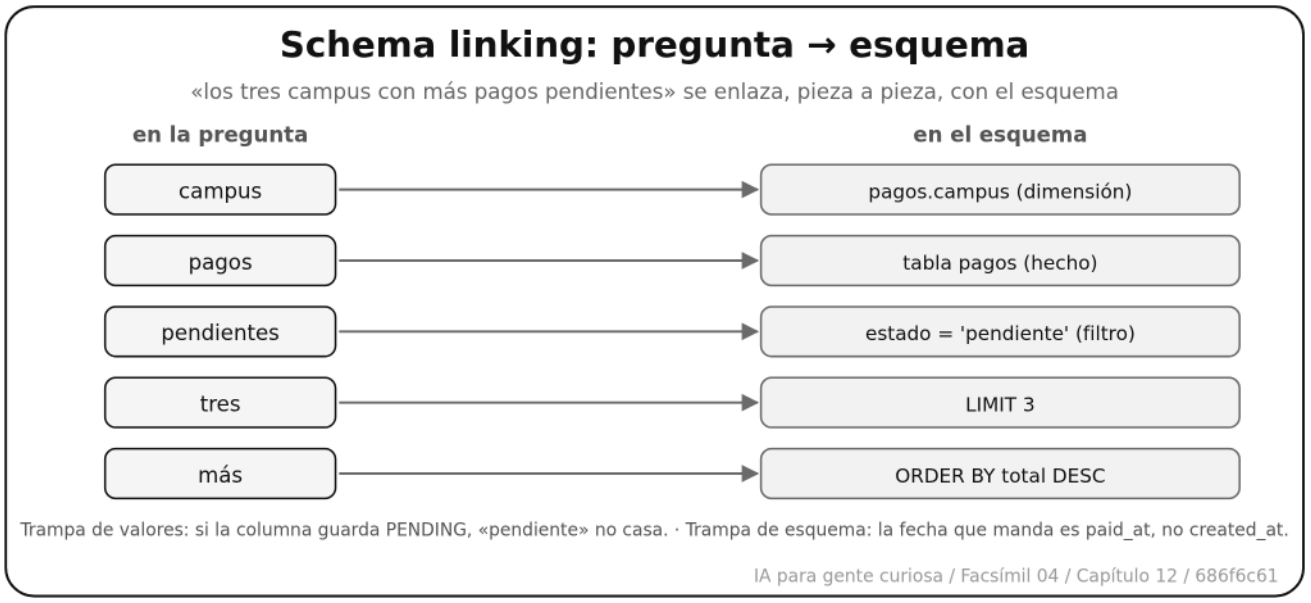
En bases pequeñas se puede meter todo el esquema en el prompt. En bases grandes no. Hay que recuperar el esquema como se recuperan documentos: por dominio, por nombres, por descripciones, por consultas de ejemplo y por permisos.

### Recuperar el subesquema relevante

Conviene insistir en esto porque es el cuello de botella real: **la mayoría de los fallos de Text-to-SQL no son de sintaxis SQL, son de schema linking**. El modelo escribe SQL válido, pero sobre la tabla o la columna equivocada. Por eso, con un esquema de cientos de tablas, el trabajo de ingeniería no es «mejorar el prompt», sino **recuperar el subesquema correcto** antes de generar, igual que un RAG recupera los chunks correctos. Las señales que ayudan a ese retrieval del esquema:

| SEÑAL | QUÉ APORTA |
|---|---|
| Nombres de tablas y columnas | El emparejamiento más directo, pero traicionero: `paid_at` frente a `created_at`. |
| Descripciones y comentarios | Lo que el nombre no dice; es donde vive la semántica de negocio. |
| Valores de ejemplo | Resuelve el problema de los valores: si la columna `estado` contiene `PENDING`, el modelo no inventará `pendiente`. |
| Claves y relaciones | Qué tablas se unen y por qué columna, para no inventar joins. |
| Consultas anteriores | Ejemplos parecidos que ya funcionaron (few-shot dirigido). |

Esta dificultad es precisamente lo que miden los benchmarks académicos de Text-to-SQL, que evalúan sobre esquemas que el modelo no ha visto antes.[9] La regla de ingeniería: trata el esquema como un corpus que se recupera, no como un texto fijo que se pega; y mide el linking por separado, porque es donde se gana o se pierde la batalla.




## La capa semántica como contrato

La capa semántica es el lugar donde una organización deja de discutir cada vez qué significa una métrica. No es una frase bonita en un prompt. Es un contrato versionado entre negocio, ingeniería y análisis.

Una definición mínima de métrica debería responder:

| PREGUNTA | EJEMPLO PARA `INGRESO_NETO` |
|---|---|
| ¿Qué entidad mide? | Pagos confirmados. |
| ¿Qué columna numérica usa? | `pagos.importe`. |
| ¿Qué estados entran? | Solo `pagado`. |
| ¿Qué estados salen? | `pendiente`, `anulado`, `devuelto`, `prueba`. |
| ¿Qué fecha manda? | `fecha_pago`, no `fecha_creacion`. |
| ¿Qué dimensiones permite? | `campus`, `titulacion`, `mes`. |
| ¿Quién puede verla? | Roles de análisis y dirección académica. |
| ¿Cómo se prueba? | Casos con resultado esperado. |

Esa misma definición puede expresarse de muchas formas: YAML, dbt metrics, una tabla de metadatos, una API interna o una vista SQL. Lo importante es que el modelo no invente la definición cada vez.

```
metric: ingreso_neto
entity: pagos
expression: SUM(importe)
filters:
  - estado = 'pagado'
time_dimension: fecha_pago
allowed_dimensions:
  - campus
  - titulacion
  - mes
blocked_columns:
  - dni
  - email_personal
owner: equipo_datos_matricula
```

Con esa capa, una pregunta como “ingresos por campus en marzo” no empieza desde cero. El sistema sabe que “ingresos” apunta a `ingreso_neto`, que marzo debe filtrarse con `fecha_pago`, que el grupo permitido es `campus` y que algunas columnas ni siquiera deben entrar en el contexto.

Para una persona curiosa, la idea es esta: una base de datos guarda datos, pero no siempre guarda significado. La capa semántica añade significado compartido. Para un ingeniero, añade algo igual de importante: reduce libertad donde la libertad produce errores.

---

## Herramientas de datos: no todo es Text-to-SQL

Text-to-SQL es una herramienta, pero no la única. A veces conviene no dejar que el modelo genere SQL libre, sino exponer operaciones más estrechas.

| TIPO DE HERRAMIENTA | QUÉ HACE | CUÁNDO USARLA | EJEMPLO |
|---|---|---|---|
| Text-to-SQL libre | Genera consultas nuevas. | Análisis exploratorio con validación fuerte. | “Agrupa pagos por campus y mes”. |
| Plantilla parametrizada | Rellena parámetros de una consulta fija. | Métricas críticas y repetibles. | `campus`, `fecha_inicio`, `fecha_fin`. |
| Stored procedure | Llama a una función definida en la base. | Regla compleja y estable. | `calcular_morosidad(campus, mes)`. |
| Semantic layer | Consulta métricas y dimensiones declaradas. | BI, reporting y definiciones de negocio. | `ingreso_neto` por `campus`. |
| DataFrame tool | Opera sobre tablas en memoria. | Exploración local, CSV, notebooks. | Pandas, Polars, DuckDB. |
| Chart tool | Convierte resultados en gráfico. | Cuando la salida natural es visual. | Barras por campus. |
| Data quality tool | Comprueba nulos, duplicados o rangos. | Antes de confiar en una respuesta. | “¿Hay importes negativos?”. |

La pregunta de arquitectura no es “¿puedo generar SQL?”. La pregunta buena es “¿qué superficie de datos quiero exponer?”. Cuanto más abierta sea la herramienta, más validación necesita.

## Contrato de una herramienta SQL

Una herramienta SQL no debería aceptar una cadena cualquiera. Debería tener un contrato que obligue a declarar intención, dominio, límites y formato de respuesta.

```
{
  "name": "consultar_datos",
  "input": {
    "question": "Pagos pendientes por campus en marzo",
    "domain": "matricula",
    "sql": "SELECT campus, SUM(importe) AS total FROM pagos ...",
    "dialect": "sqlite",
    "max_rows": 50,
    "timeout_ms": 1500,
    "purpose": "analisis_agregado"
  },
  "output": {
    "columns": ["campus", "total"],
    "rows": [["Norte", 19320.0]],
    "row_count": 1,
    "elapsed_ms": 24,
    "trace_id": "f4c12-001"
  },
  "errors": [
    "tabla_no_permitida",
    "consulta_no_select",
    "demasiadas_filas",
    "timeout",
    "sql_invalido"
  ]
}
```


Cada campo tiene una razón:

| CAMPO | QUÉ CONTROLA | POR QUÉ IMPORTA |
|---|---|---|
| `domain` | Área funcional. | Evita mezclar tablas sin contexto. |
| `sql` | Consulta candidata. | Debe poder validarse y registrarse. |
| `dialect` | Motor SQL esperado. | `LIMIT`, fechas y funciones cambian entre motores. |
| `max_rows` | Filas máximas. | Protege coste y evita respuestas inmanejables. |
| `timeout_ms` | Tiempo máximo. | Una mala consulta no debe bloquear el sistema. |
| `purpose` | Uso declarado. | No es igual explorar que cerrar un informe. |
| `trace_id` | Identificador de ejecución. | Permite reproducir y auditar. |

La herramienta puede estar detrás de OpenAI Function Calling, de un agente de LangChain, de LlamaIndex, de una API propia o de un servicio interno. El principio no cambia: el modelo propone argumentos estructurados, pero el servidor valida y ejecuta.

## Permisos, datos sensibles y trazas

Una herramienta de datos tiene tres identidades distintas y conviene no mezclarlas:

| IDENTIDAD | QUÉ REPRESENTA | ERROR TÍPICO |
|---|---|---|
| Persona usuaria | Quien hace la pregunta. | Darle acceso por el rol técnico del servidor. |
| Modelo | Quien propone la consulta. | Tratar su SQL como decisión autorizada. |
| Servicio de datos | Quien ejecuta. | Conectar con permisos demasiado amplios. |

El modelo no debería tener permisos. Quien tiene permisos es el servicio que recibe una petición, revisa el rol de la persona, aplica reglas y ejecuta una consulta limitada. Esa separación evita que una respuesta dependa de una frase del prompt.

En una arquitectura seria, pondría estos controles:

| CONTROL | QUÉ PROTEGE | EJEMPLO CONCRETO |
|---|---|---|
| Conexión read-only | Evita modificar datos. | Usuario SQL sin `INSERT`, `UPDATE`, `DELETE` ni DDL. |
| Réplica de lectura | Aísla producción. | Consultar una réplica o almacén analítico. |
| Row-level security | Limita filas por rol. | Un campus solo ve sus alumnos. |
| Column allowlist | Limita columnas visibles. | Exponer `campus`, no `dni` ni `email_personal`. |
| Query timeout | Evita consultas largas. | Cortar a 1.500 ms en exploración. |
| Row limit | Controla volumen de salida. | Máximo 100 filas por respuesta. |
| Redacción de logs | Evita guardar datos innecesarios. | Registrar SQL y hash de usuario, no tabla completa. |
| Trazabilidad | Permite reproducir. | `trace_id`, versión de esquema, modelo y validador. |

La traza no es vigilancia decorativa. Es el expediente técnico de la respuesta. Si alguien pregunta “¿de dónde salió este número?”, necesitas poder reconstruir pregunta, usuario, rol, tablas candidatas, SQL generado, reglas aplicadas, resultado, latencia y versión del sistema.

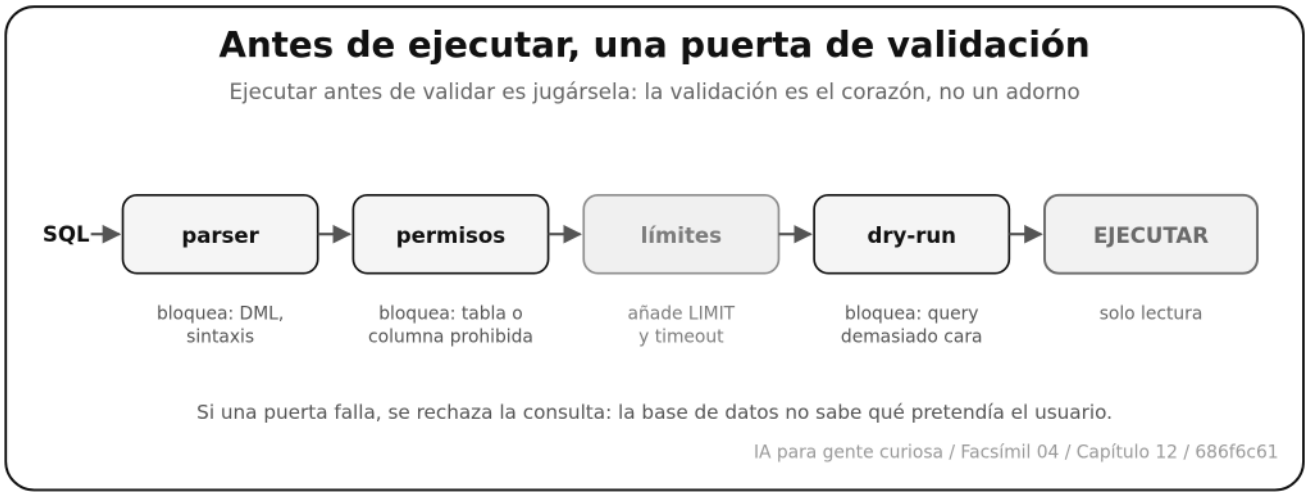
---

## Validación antes de ejecutar

La validación no es un adorno. Es el corazón del sistema. Un validador mínimo debería revisar:

| CONTROL | QUÉ COMPRUEBA | EJEMPLO DE REGLA |
|---|---|---|
| Tipo de consulta | Solo lectura. | Aceptar únicamente `SELECT` o `WITH ... SELECT`. |
| Tablas permitidas | Superficie acotada. | `pagos`, `alumnos`, `campus`. |
| Columnas permitidas | Evitar columnas fuera de contrato. | No exponer `dni` si no hace falta. |
| Límite de filas | Salida manejable. | Añadir o exigir `LIMIT 100`. |
| Timeout | Coste temporal. | Cortar a 1,5 segundos. |
| Dialecto | Sintaxis correcta. | SQLite no es BigQuery. |
| Agregación | Preguntas agregadas devuelven agregados. | `SUM`, `COUNT`, `AVG`. |
| Filtros de usuario | Permisos por rol. | Campus permitido por sesión. |
| Explicación | Respuesta basada en resultado. | No resumir columnas que no salieron. |

Herramientas como SQLGlot ayudan a parsear, inspeccionar y transpilar SQL entre dialectos.[10] Motores embebidos como DuckDB son útiles para análisis local, CSV, Parquet o notebooks, porque permiten ejecutar SQL desde Python sin levantar un servidor externo.[11]




## Query plan: cuando una consulta correcta no cabe

Una consulta puede ser correcta y aun así no ser aceptable. Si tarda treinta segundos, bloquea recursos o escanea una tabla enorme para devolver tres filas, el problema no es solo del modelo: es del sistema que no miró el plan.

El `query plan` es la explicación interna que calcula el motor antes de ejecutar. Según el motor puede verse con `EXPLAIN`, `EXPLAIN ANALYZE`, perfiles de consulta o dry-run. Los nombres cambian, pero la pregunta es la misma: ¿qué tendrá que hacer la base para responder?

```
EXPLAIN
SELECT campus, SUM(importe) AS total_pendiente
FROM pagos
WHERE estado = 'pendiente'
GROUP BY campus
ORDER BY total_pendiente DESC
LIMIT 3;
```

Al leer un plan, no hace falta entender todo desde el primer día. Empieza por estas señales:

| SEÑAL | QUÉ SIGNIFICA | POR QUÉ IMPORTA |
|---|---|---|
| Full scan | Recorrer muchas filas de una tabla. | Puede ser normal en tablas pequeñas, caro en tablas grandes. |
| Index scan | Localizar filas con un índice. | Suele ayudar si el filtro es selectivo. |
| Cardinalidad estimada | Filas que el motor cree que pasarán. | Si estima mal, elige planes pobres. |
| Join strategy | Forma de unir tablas. | Nested loop, hash join o merge join tienen costes distintos. |
| Sort | Ordenación intermedia. | `ORDER BY` sobre muchas filas puede ser caro. |
| Temporary spill | Datos intermedios fuera de memoria. | Señal de presión de memoria o consulta pesada. |

Un índice no resuelve todo. Acelera ciertas búsquedas a cambio de ocupar espacio y complicar escrituras. Si filtras mucho por `estado` y `fecha`, un índice puede ayudar:

```
CREATE INDEX idx_pagos_estado_fecha
ON pagos (estado, fecha);
```

Pero si casi todos los pagos están en `estado = 'pendiente'`, ese índice quizá no aporta mucho, porque el filtro no reduce bastante. Esta idea se llama selectividad: un filtro útil descarta muchas filas.

Para Text-to-SQL, el plan sirve como control previo:

| PREGUNTA TÉCNICA | DECISIÓN DEL SISTEMA |
|---|---|
| ¿Escanea más filas de las permitidas? | Pedir aclaración, añadir filtro o bloquear. |
| ¿Usa tablas fuera del dominio? | Rechazar y pedir reformulación. |
| ¿Ordena millones de filas sin agregación previa? | Proponer una consulta agregada. |
| ¿No hay índice para el filtro principal? | Avisar de latencia o derivar a informe offline. |

## Dialecto SQL: el mismo pedido cambia por motor

SQL tiene una gramática común, pero cada motor añade funciones, tipos y límites. Un sistema que genera SQL debe saber para qué motor escribe. SQLite no es PostgreSQL, PostgreSQL no es BigQuery, BigQuery no es Snowflake.

| NECESIDAD | SQLITE | POSTGRESQL | BIGQUERY | SNOWFLAKE |
|---|---|---|---|---|
| Limitar filas | `LIMIT 10` | `LIMIT 10` | `LIMIT 10` | `LIMIT 10` |
| Mes de una fecha | `strftime('%Y-%m', fecha)` | `date_trunc('month', fecha)` | `DATE_TRUNC(fecha, MONTH)` | `DATE_TRUNC('MONTH', fecha)` |
| Concatenar texto | `a | | b` | `a |
| Fecha actual | `date('now')` | `CURRENT_DATE` | `CURRENT_DATE()` | `CURRENT_DATE()` |
| Muestra aproximada | Limitado | `TABLESAMPLE` | `TABLESAMPLE SYSTEM` | `SAMPLE` |

No hace falta memorizar todos los dialectos. Lo importante es no mezclar. Si el contrato dice `dialect: "sqlite"`, el generador, el parser, los ejemplos y el validador deben hablar SQLite. Si el almacén real es BigQuery, conviene validar con BigQuery o con un parser que entienda sus particularidades.

Esta es una razón más para no pegar ejemplos al azar en el prompt. Un ejemplo de PostgreSQL puede enseñar al modelo una función que luego falla en BigQuery. Los ejemplos son datos de entrenamiento local para la consulta que estás a punto de generar; si están mal elegidos, orientan mal.

## Cómo elegir arquitectura

La solución correcta depende del riesgo, del tamaño del esquema y de lo repetible que sea la pregunta.

| SITUACIÓN | ARQUITECTURA RECOMENDADA | POR QUÉ |
|---|---|---|
| Métrica crítica y estable. | Plantilla parametrizada. | Menos libertad, más confianza. |
| Exploración interna con datos agregados. | Text-to-SQL con validación y trazas. | Permite flexibilidad controlada. |
| BI con definiciones compartidas. | Semantic layer + modelo. | Las métricas viven fuera del prompt. |
| CSV local o notebook. | DuckDB/DataFrame tool. | Iteración rápida y entorno cerrado. |
| Esquema enorme. | Retrieval de esquema + examples RAG. | No cabe todo el catálogo. |
| Varias bases y documentos. | Router + herramientas especializadas. | No todo debe ir por SQL. |
| Preguntas repetidas de negocio. | Stored procedures o vistas. | La lógica queda versionada. |

Mi regla práctica: si la pregunta puede romper un informe importante, no empieces con SQL libre. Empieza con métrica declarada, plantilla o vista. Usa Text-to-SQL libre para exploración, no para convertir cada pregunta de negocio en una consulta nueva sin revisión.

## Coste, latencia y contexto

Text-to-SQL parece barato porque la salida es corta. Pero el contexto puede crecer muchísimo: documentación de tablas, columnas, ejemplos, métricas, permisos, dialecto y trazas anteriores.

El contexto que entra al modelo es la suma de todo lo que tiene que ver para escribir SQL correcto, y crece rápido:

$$T_{\text{ctx}} = T_{\text{pregunta}} + T_{\text{schema}} + T_{\text{docs}} + T_{\text{ejemplos}} + T_{\text{politicas}}$$

| SÍMBOLO | SIGNIFICADO | EJEMPLO |
|---|---|---|
| $T_{\text{ctx}}$ | Tokens totales de contexto. | 6.400 tokens. |
| $T_{\text{pregunta}}$ | Pregunta del usuario. | 30 tokens. |
| $T_{\text{schema}}$ | Tablas, columnas y claves incluidas. | 2.500 tokens. |
| $T_{\text{docs}}$ | Documentación de negocio. | 1.200 tokens. |
| $T_{\text{ejemplos}}$ | Consultas parecidas. | 1.800 tokens. |
| $T_{\text{politicas}}$ | Permisos y reglas. | 870 tokens. |

La latencia total tampoco es solo la generación del LLM: es la suma de toda la cadena, desde enrutar la pregunta hasta redactar la respuesta:

$$L_{\text{total}} = L_{\text{router}} + L_{\text{schema}} + L_{\text{LLM}} + L_{\text{validacion}} + L_{\text{db}} + L_{\text{resumen}}$$

| SÍMBOLO | SIGNIFICADO | EJEMPLO |
|---|---|---|
| $L_{\text{schema}}$ | Tiempo de recuperar esquema relevante. | 80 ms. |
| $L_{\text{LLM}}$ | Tiempo de generar SQL o plan. | 1.200 ms. |
| $L_{\text{validacion}}$ | Parser, reglas y dry-run. | 90 ms. |
| $L_{\text{db}}$ | Ejecución en base de datos. | 300 ms. |
| $L_{\text{resumen}}$ | Redacción final. | 700 ms. |

Y el coste por pregunta suma todo lo que se paga, no solo los tokens del modelo:

$$C = C_{\text{modelo}} + C_{\text{db}} + C_{\text{observabilidad}} + C_{\text{mantenimiento}}$$

| SÍMBOLO | SIGNIFICADO | EJEMPLO |
|---|---|---|
| $C_{\text{modelo}}$ | Tokens de entrada y salida del LLM. | Factura del proveedor por consulta. |
| $C_{\text{db}}$ | Cómputo de ejecutar la consulta en la base de datos. | Warehouse que cobra por consulta o por cómputo escaneado. |
| $C_{\text{observabilidad}}$ | Trazas, logs y métricas de cada pregunta. | Almacenamiento y herramienta de observabilidad. |
| $C_{\text{mantenimiento}}$ | Trabajo humano de mantener el sistema. | Documentar columnas, versionar métricas, revisar fallos. |

La parte invisible suele ser $C_{\text{mantenimiento}}$: documentar columnas, versionar métricas, revisar consultas fallidas, actualizar ejemplos y controlar cambios de esquema. Un Text-to-SQL no se paga una vez; se paga cada vez que el esquema cambia.

Juntando las tres cuentas en una pregunta concreta, con los valores de los ejemplos de arriba:

| MAGNITUD | CUENTA | RESULTADO |
|---|---|---|
| Contexto | $30 + 2500 + 1200 + 1800 + 870$ | 6 400 tokens |
| Latencia | $20 + 80 + 1200 + 90 + 300 + 700$ ms (router, esquema, LLM, validación, db, resumen) | ≈ 2,4 s |
| Coste del modelo | 6 400 tokens de entrada + 150 de salida, a tarifas ilustrativas de 2,00/8,00 por millón | ≈ 0,014 por pregunta |

La lectura de ingeniería: el coste del modelo (0,014) parece ridículo, pero **el contexto es el 98 % de los tokens**, y de él la mayor parte es el esquema y los ejemplos. Por eso la palanca real de coste y latencia no es el modelo más barato, sino **recuperar menos esquema** (volvemos al schema linking): cada tabla que no metes en el prompt baja las tres cuentas a la vez.

## Evaluar Text-to-SQL

La evaluación clásica compara SQL generado contra SQL esperado. Eso ayuda, pero no basta. Dos consultas distintas pueden devolver el mismo resultado; dos consultas parecidas pueden divergir en casos frontera.

| MÉTRICA | QUÉ MIDE | CUIDADO |
|---|---|---|
| Exact match | Si el SQL coincide con el esperado. | Penaliza consultas equivalentes escritas distinto. |
| Execution accuracy | Si produce el resultado correcto. | Puede acertar por casualidad en datos pequeños. |
| Result-set match | Si filas y columnas coinciden. | Hay que controlar orden, tipos y redondeo. |
| Component match | Si `SELECT`, `WHERE`, `JOIN`, `GROUP BY` están bien. | Útil para depurar. |
| Schema-link accuracy | Si eligió tablas y columnas correctas. | Necesita anotación o revisión. |
| Permission pass rate | Si respeta permisos y límites. | No basta medir precisión. |
| Query cost | Coste estimado o real de ejecución. | Una query correcta puede ser inviable. |
| Latencia P95 | Tiempo para el 95% de consultas. | La media oculta colas lentas. |
| Clarification rate | Cuándo pide aclaración. | Preguntar puede ser mejor que inventar. |

Estas métricas no son invento nuestro: vienen de los benchmarks académicos de Text-to-SQL. **Spider** estandarizó la evaluación sobre esquemas no vistos y popularizó dos medidas: el *exact match* (componente a componente) y la *execution accuracy* (mismas filas que el SQL de oro).[12] **BIRD** dio un paso hacia el mundo real: bases sucias y grandes, valores con ruido, y una métrica de *valid efficiency score* que penaliza consultas correctas pero carísimas.[13] La lección que se traslada a tu sistema: **execution accuracy importa más que exact match** (lo que cuenta es la cifra correcta, no que el SQL sea idéntico), pero ninguna métrica pública sustituye a un dataset con tu esquema, tus valores y tus permisos.

Un dataset propio debería tener:

| CAMPO | EJEMPLO |
|---|---|
| `question` | “Pagos pendientes por campus en marzo”. |
| `user_role` | `analista_matricula`. |
| `allowed_tables` | `pagos`, `alumnos`, `campus`. |
| `gold_sql` | Consulta esperada o plantilla. |
| `expected_result` | Filas esperadas. |
| `must_not_use` | Columnas que no proceden. |
| `notes` | Definición de negocio relevante. |

En producción mediría, como mínimo:

| CAPA | PREGUNTA DE EVALUACIÓN |
|---|---|
| Clasificador | ¿Detectó que era una pregunta de datos? |
| Selector de esquema | ¿Incluyó las tablas necesarias y excluyó ruido? |
| Generador | ¿Produjo SQL válido para el dialecto? |
| Validador | ¿Bloqueó lo que debía bloquear? |
| Ejecución | ¿Devolvió resultado correcto dentro de límite? |
| Resumen | ¿Explicó solo lo que aparece en la tabla? |
| Trazas | ¿Puedo reproducir la respuesta? |

Una forma práctica de empezar es construir un pequeño harness de evaluación. No tiene que ser perfecto. Tiene que hacer visible cuándo el sistema mejora o empeora.

```
{
  "id": "matricula-001",
  "question": "Alumnos con pago pendiente por campus en marzo",
  "role": "analista_matricula",
  "dialect": "sqlite",
  "allowed_tables": ["pagos", "alumnos"],
  "expected_sql_patterns": ["GROUP BY campus", "estado = 'pendiente'"],
  "expected_result": [
    {"campus": "Norte", "alumnos": 2},
    {"campus": "Centro", "alumnos": 1}
  ],
  "max_latency_ms": 1500,
  "max_rows": 20,
  "must_not_use": ["dni", "email_personal"],
  "review_note": "Debe contar alumnos únicos, no pagos."
}
```

En ese ejemplo no basta con que aparezca `GROUP BY campus`. El caso dice explícitamente que hay que contar alumnos únicos. Esa nota evita que una consulta con `COUNT(*)` pase por casualidad cuando los datos de prueba son pequeños.

Un harness útil debería guardar cuatro salidas:

| SALIDA | PARA QUÉ SIRVE |
|---|---|
| SQL generado | Revisar estructura y dialecto. |
| Resultado devuelto | Comparar con la tabla esperada. |
| Razón de validación | Saber si el sistema aceptó o bloqueó con criterio. |
| Traza | Reproducir el caso con misma versión de esquema y modelo. |

Y debería separar tipos de error. No es igual fallar por sintaxis que fallar por semántica:

| TIPO DE FALLO | EJEMPLO | QUÉ ARREGLAR |
|---|---|---|
| Sintaxis | Función inexistente para el dialecto. | Ejemplos y parser. |
| Schema linking | Usa `created_at` en lugar de `paid_at`. | Descripciones y selector de esquema. |
| Semántica | Cuenta pagos cuando debía contar alumnos. | Capa semántica y casos de prueba. |
| Permisos | Incluye columna no permitida. | Validador y allowlist. |
| Coste | Query correcta pero pesada. | Plan, índices, límites o vista agregada. |
| Explicación | Resume algo que no está en el resultado. | Contrato de respuesta y groundedness. |

## Cuando conviene pedir aclaración

Una buena herramienta de datos no responde siempre. A veces pregunta.

| PREGUNTA ORIGINAL | QUÉ FALTA | MEJOR RESPUESTA DEL SISTEMA |
|---|---|---|
| “Ventas del mes”. | Mes, definición de venta, región. | “¿Qué mes y qué métrica de ventas quieres usar?” |
| “Alumnos activos”. | Definición de activo. | “Puedo usar matrícula vigente o acceso reciente. ¿Cuál prefieres?” |
| “Top clientes”. | Top por ingresos, pedidos o margen. | “¿Quieres ordenar por ingresos, margen o número de pedidos?” |
| “Comparar campus”. | Periodo y métrica. | “Dime periodo y métrica principal.” |

Pedir aclaración no es fallar. En datos, muchas respuestas incorrectas nacen de contestar demasiado rápido.

## Caso completo: de pregunta a respuesta trazable

Tomemos una pregunta realista:

> “Dame los tres campus con más alumnos con pagos pendientes en marzo.”

Parece una pregunta simple. No lo es. Hay una entidad, alumnos; un evento, pagos; una condición, pendientes; una ventana temporal, marzo; una agrupación, campus; un ranking, los tres primeros.

El sistema debería recorrer algo parecido a esto:

| PASO | DECISIÓN | RESULTADO INTERMEDIO |
|---|---|---|
| 1. Intención | Es una pregunta de datos agregados. | Ruta a herramienta SQL. |
| 2. Dominio | Matrícula y pagos. | No consulta documentos generales. |
| 3. Usuario | Rol `analista_matricula`. | Puede ver agregados por campus. |
| 4. Esquema | Necesita `pagos` y quizá `alumnos`. | No carga todo el catálogo. |
| 5. Semántica | “Alumnos con pagos pendientes” cuenta alumnos únicos. | Métrica: `COUNT(DISTINCT alumno_id)`. |
| 6. Fecha | Marzo se interpreta como rango semiabierto. | `>= '2026-03-01'` y `< '2026-04-01'`. |
| 7. SQL candidato | Genera consulta agregada. | `GROUP BY campus`, `ORDER BY`, `LIMIT 3`. |
| 8. Validación | Solo lectura, tablas permitidas, límite. | Acepta o pide corrección. |
| 9. Plan | Estima filas y coste. | Bloquea si escanea demasiado. |
| 10. Ejecución | Ejecuta en réplica de lectura. | Devuelve tabla pequeña. |
| 11. Respuesta | Explica solo la tabla. | No inventa causas. |
| 12. Traza | Registra expediente. | Permite reproducir. |

Un SQL razonable podría ser:

```
SELECT
  p.campus,
  COUNT(DISTINCT p.alumno_id) AS alumnos_con_pago_pendiente
FROM pagos AS p
WHERE p.estado = 'pendiente'
  AND p.fecha >= '2026-03-01'
  AND p.fecha < '2026-04-01'
GROUP BY p.campus
ORDER BY alumnos_con_pago_pendiente DESC
LIMIT 3;
```

Fíjate en lo que no hace:

| NO HACE | POR QUÉ |
|---|---|
| No usa `COUNT(*)`. | Contaría pagos, no alumnos. |
| No usa `estado != 'pagado'`. | Metería estados no definidos. |
| No pide columnas personales. | La pregunta solo necesita agregados. |
| No devuelve filas individuales. | La salida pedida es un ranking. |
| No explica causas. | La consulta no investigó causas, solo conteos. |

La respuesta humana debería sonar así:

> “Con la definición de pagos pendientes como `estado = 'pendiente'` y tomando marzo como `[2026-03-01, 2026-04-01)`, los tres campus con más alumnos únicos con pagos pendientes son Norte, Centro y Sur. La consulta usa datos agregados y no incluye información personal.”

Y la traza técnica podría guardar:

```
{
  "trace_id": "f4c12-demo-023",
  "question": "Dame los tres campus con más alumnos con pagos pendientes en marzo",
  "route": "sql_tool",
  "role": "analista_matricula",
  "dialect": "sqlite",
  "tables": ["pagos"],
  "metric": "count_distinct_alumno_id",
  "validated": true,
  "checks": ["read_only", "allowed_tables", "row_limit", "date_range"],
  "row_count": 3,
  "elapsed_ms": 31
}
```


Esta traza no es para enseñarla entera a la persona que pregunta. Es para que el equipo pueda auditar, depurar y mejorar el sistema.

---

## Soluciones de terceros y piezas habituales

Hay herramientas ya hechas, pero conviene saber qué problema resuelve cada una. No todas sustituyen una arquitectura propia.

| PIEZA | QUÉ APORTA | QUÉ REVISARÍA ANTES DE USARLA |
|---|---|---|
| LangChain SQL Agent | Flujo agentic para inspeccionar esquema, generar, revisar y ejecutar SQL. | Permisos de conexión, trazas, límites y revisión humana. |
| LlamaIndex NL SQL | Query engines para lenguaje natural sobre tablas SQL. | Qué esquema entra, cómo controla ejecución y cómo registra fuentes. |
| Vanna | Enfoque natural language -> SQL -> respuestas con permisos y componentes de UI. | El repositorio público aparece archivado desde marzo de 2026; revisaría mantenimiento y versión usada.[14] |
| SQLGlot | Parser, AST y transpiler SQL. | Cobertura del dialecto y reglas propias de validación. |
| DuckDB | Motor local para análisis, CSV y Parquet desde Python. | Memoria, tamaño de datos y diferencias frente al motor de producción. |
| dbt / capa semántica | Métricas y transformaciones versionadas. | Quién mantiene definiciones y cómo se exponen al modelo. |
| BI tradicional | Dashboards y métricas curadas. | Qué preguntas quedan fuera del dashboard. |

Una buena arquitectura puede mezclar varias: LangChain o LlamaIndex para orquestar, SQLGlot para validar, DuckDB para prototipos locales, una capa semántica para métricas y una API propia para permisos.

## En el día a día

En el día a día, Text-to-SQL aparece cuando la respuesta no está escrita en ningún documento, sino que hay que **calcularla** sobre datos vivos: «¿cuántos alumnos con pagos pendientes hay en el campus Norte este mes?». Ningún chunk contiene ese número; sale de una consulta. Es lo que permite que una persona de negocio pregunte en lenguaje natural y obtenga una cifra de la base de datos sin saber SQL. En un producto, es el puente entre el RAG (que recupera texto) y el estado exacto del sistema (que vive en tablas).

La parte delicada es que aquí un error no se nota como un texto raro: se nota como un **número equivocado que parece correcto**. Una consulta puede ejecutar sin error y calcular otra cosa (contar pagos en vez de alumnos, sumar una columna que no era), y nadie lo ve hasta que alguien toma una decisión con ese dato. Por eso Text-to-SQL es, sobre todo, un problema de validación y de semántica, no de generar SQL bonito.

## Por qué debería importarte

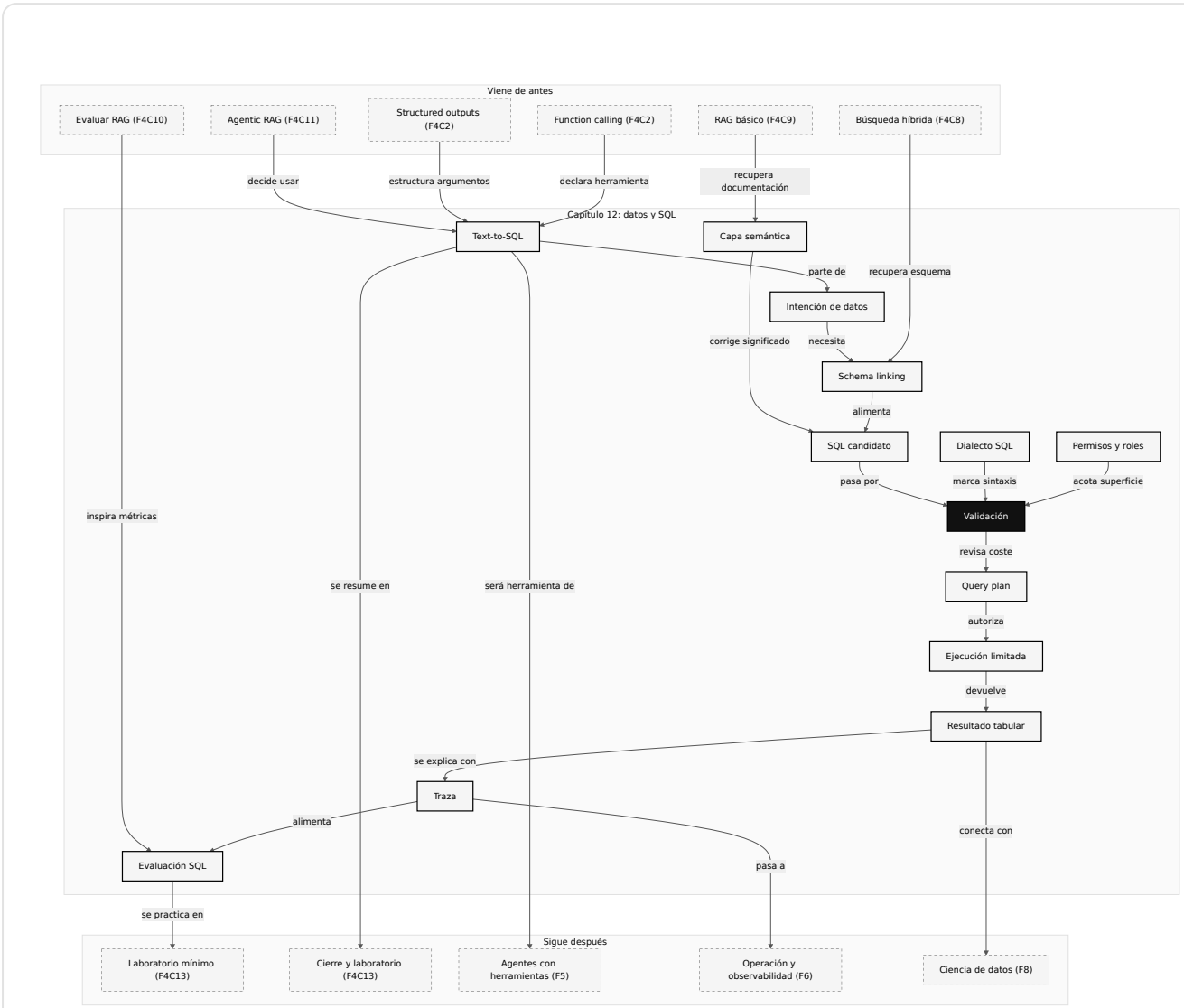
Porque es donde la IA toca los datos de verdad, y donde un fallo cuesta más: una respuesta de RAG mal fundamentada es un texto dudoso; un número mal calculado es una decisión mal tomada. Si entiendes la cadena (esquema, semántica, validación, permisos, ejecución y traza), sabes por qué un sistema responde un número y puedes defenderlo o corregirlo. Sin esa cadena, tienes un generador de SQL que a veces acierta y nunca sabes cuándo.

También importa porque conecta dos mundos que la mayoría de la gente mantiene separados: el lenguaje natural y el modelo de datos. Saber traducir «ingresos del trimestre» a la métrica exacta, con su definición versionada y sus permisos, es una habilidad de ingeniería que no caduca, use el motor que use por debajo.

## Dónde solía tropezar yo

| ERROR | POR QUÉ ES UN ERROR | ANTÍDOTO |
|---|---|---|
| **Confundir SQL válido con respuesta correcta** | Una consulta puede ejecutar y aun así calcular otra cosa. | Evaluar resultado, tablas usadas y definición de negocio. |
| **Meter todo el esquema en el prompt** | Con esquemas grandes sube el coste y entra ruido. | Recuperar solo tablas, columnas y ejemplos relevantes. |
| **Olvidar la capa semántica** | “Ingresos”, “activo” o “pendiente” no significan lo mismo en cada empresa. | Declarar métricas y definiciones fuera del prompt. |
| **Contar filas cuando quería entidades** | `COUNT(*)` puede contar pagos, no alumnos. | Revisar cardinalidad y usar `COUNT(DISTINCT ...)` cuando proceda. |
| **Ignorar el plan de consulta** | Una query correcta puede ser demasiado cara. | Revisar `EXPLAIN`, cardinalidad, índices y límites. |
| **Mezclar dialectos** | Una función válida en un motor puede fallar en otro. | Pasar `dialect` en el contrato y validar contra ese motor. |
| **Ejecutar antes de validar** | La base de datos no sabe qué pretendía el usuario. | Parsear, limitar, revisar permisos y hacer dry-run. |
| **Medir solo exact match** | SQL distinto puede producir el mismo resultado correcto. | Combinar execution accuracy, result-set match y revisión por componentes. |
| **No registrar trazas** | No puedes explicar por qué salió un número. | Guardar pregunta, esquema, SQL, resultados, latencia y versión. |

# Cómo encaja todo



# Vocabulario aprendido

| TÉRMINO | DEFINICIÓN |
|---|---|
| **Text-to-SQL** | Traducción controlada de una pregunta humana a una consulta SQL ejecutable. |
| **SQL** | Lenguaje declarativo para consultar y transformar datos relacionales. |
| **Tabla** | Conjunto de filas y columnas. |
| **Fila** | Registro individual dentro de una tabla. |
| **Columna** | Atributo de una tabla, como fecha, estado o importe. |
| **Clave primaria** | Columna o conjunto de columnas que identifica una fila única. |
| **Clave foránea** | Columna que conecta una tabla con otra. |
| **JOIN** | Unión entre tablas mediante una relación, normalmente una clave. |
| **Cardinalidad** | Número aproximado de filas que participan en una operación. |
| **Índice** | Estructura que ayuda a encontrar filas sin recorrer toda la tabla. |
| **Schema linking** | Vincular palabras de la pregunta con tablas, columnas, claves y valores. |
| **Dialecto SQL** | Variante de SQL de un motor concreto. |
| **Semantic layer** | Capa de métricas y reglas de negocio compartidas. |
| **Dry-run** | Comprobación previa de una consulta antes de ejecutarla plenamente. |
| **Query plan** | Plan interno de ejecución calculado por la base de datos. |
| **Read-only** | Conexión que solo permite lectura. |
| **Row-level security** | Regla que limita qué filas puede ver cada rol o persona. |
| **Execution accuracy** | Métrica que evalúa si el resultado producido por la consulta es correcto. |
| **Result-set match** | Comparación entre resultado esperado y resultado devuelto. |
| **Row limit** | Límite máximo de filas que puede devolver una consulta. |
| **Traza SQL** | Registro de pregunta, SQL, usuario, tablas, tiempo, resultado y errores. |

## Antes de pasar página

- [ ] ¿Puedo explicar por qué Text-to-SQL no es acceso libre a una base de datos? (Si no, vuelve a «Qué no es Text-to-SQL».)
- [ ] ¿Sé leer una consulta con `SELECT`, `FROM`, `WHERE`, `JOIN`, `GROUP BY`, `ORDER BY` y `LIMIT`? (Si no, vuelve a «SQL desde cero, pero sin rebajarlo».)
- [ ] ¿Puedo explicar qué es schema linking con un ejemplo? (Si no, vuelve a «El problema real: esquema, semántica y valores».)
- [ ] ¿Sé distinguir entidad, evento, clave primaria y clave foránea? (Si no, vuelve a «SQL desde cero, pero sin rebajarlo».)
- [ ] ¿Puedo detectar cuándo un `JOIN` duplica filas? (Si no, vuelve a «Errores SQL que cambian la historia».)
- [ ] ¿Sé por qué `COUNT(*)` y `COUNT(DISTINCT ...)` no responden a lo mismo? (Si no, vuelve a «Errores SQL que cambian la historia».)
- [ ] ¿Sé distinguir SQL libre, plantilla parametrizada, stored procedure y semantic layer? (Si no, vuelve a «Herramientas de datos: no todo es Text-to-SQL».)
- [ ] ¿Puedo diseñar un contrato mínimo para una herramienta SQL? (Si no, vuelve a «Contrato de una herramienta SQL».)
- [ ] ¿Sé qué validaciones haría antes de ejecutar una consulta? (Si no, vuelve a «Validación antes de ejecutar».)
- [ ] ¿Sé leer las señales básicas de un query plan? (Si no, vuelve a «Query plan: cuando una consulta correcta no cabe».)
- [ ] ¿Sé por qué el dialecto SQL debe viajar en el contrato? (Si no, vuelve a «Dialecto SQL: el mismo pedido cambia por motor».)
- [ ] ¿Puedo calcular por qué el contexto crece con esquema, documentación y ejemplos? (Si no, vuelve a «Coste, latencia y contexto».)
- [ ] ¿Sé diferenciar exact match, execution accuracy y result-set match? (Si no, vuelve a «Evaluar Text-to-SQL».)
- [ ] ¿Sé qué revisar en la traza de una consulta Text-to-SQL para confiar en el resultado? (Si no, repasa la sección correspondiente.)

## En resumen

| IDEA FUERZA | DETALLE |
|---|---|
| Text-to-SQL no es una query suelta. | Es una cadena con intención, esquema, SQL, validación, ejecución, resultado y traza. |
| El esquema manda. | Si el sistema no entiende tablas, columnas, claves y valores, generará consultas frágiles. |
| La semántica evita números falsos. | Las métricas de negocio deben estar definidas fuera del prompt. |
| La cardinalidad cambia respuestas. | Contar filas, eventos o entidades no es lo mismo. |
| El plan importa. | Una consulta correcta puede ser demasiado cara para ejecutarse en vivo. |
| La validación es obligatoria. | Solo lectura, límites, permisos, dialecto, timeout y trazas antes de ejecutar. |
| La evaluación mira resultados. | Exact match ayuda, pero execution accuracy y result-set match son centrales. |

## Para saber más


DuckDB. (2026). *Python API*. Documentación oficial

LangChain. (2026). *Build a SQL agent*. Documentación oficial

Lei, F. y otros (2024). *Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows*. arXiv

Li, J. y otros (2023). *Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs*. arXiv

LlamaIndex. (2026). *NL SQL table query engine*. Documentación oficial

OpenAI. (2026). *Function calling*. Documentación oficial

OpenAI. (2026). *Structured model outputs*. Documentación oficial

Scholak, T., Schucher, N. y Bahdanau, D. (2021). *PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models*. *Proceedings of EMNLP*, 9895-9901. DOI

SQLGlot. (2026). *Python SQL parser and transpiler*. Documentación

Vanna AI. (2026). *Vanna 2.0: Turn Questions into Data Insights*. GitHub

Wang, B., Shin, R., Liu, X., Polozov, O. y Richardson, M. (2020). *RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers*. *Proceedings of ACL*, 7567-7578. DOI

Yu, T. y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. *Proceedings of EMNLP*, 3911-3921. ACL Anthology

---

## Notas

1. Yu, T. y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. *Proceedings of EMNLP*, 3911-3921. ACL Anthology. ↩

2. Wang, B., Shin, R., Liu, X., Polozov, O. y Richardson, M. (2020). *RAT-SQL: Relation-Aware Schema Encoding and Linking for Text-to-SQL Parsers*. *Proceedings of ACL*, 7567-7578. DOI. ↩

3. Scholak, T., Schucher, N. y Bahdanau, D. (2021). *PICARD: Parsing Incrementally for Constrained Auto-Regressive Decoding from Language Models*. *Proceedings of EMNLP*, 9895-9901. DOI. ↩

4. Li, J. y otros (2023). *Can LLM Already Serve as A Database Interface? A BIg Bench for Large-Scale Database Grounded Text-to-SQLs*. arXiv. ↩

5. Lei, F. y otros (2024). *Spider 2.0: Evaluating Language Models on Real-World Enterprise Text-to-SQL Workflows*. arXiv. ↩

6. LangChain. (2026). *Build a SQL agent*. Documentación oficial. Consultado el 26 de mayo de 2026. ↩

7. LlamaIndex. (2026). *NL SQL table query engine*. Documentación oficial. Consultado el 26 de mayo de 2026. ↩

8. OpenAI. (2026). *Function calling*. Documentación oficial. OpenAI. (2026). *Structured model outputs*. Documentación oficial. Consultado el 26 de mayo de 2026. ↩

9. Tao Yu y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. EMNLP. https://arxiv.org/abs/1809.08887. ↩

10. SQLGlot. (2026). *Python SQL parser and transpiler*. Documentación. Consultado el 26 de mayo de 2026. ↩

11. DuckDB. (2026). *Python API*. Documentación oficial. Consultado el 26 de mayo de 2026. ↩

.2. Tao Yu y otros (2018). *Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task*. EMNLP. https://arxiv.org/abs/1809.08887. ↩

.3. Jinyang Li y otros (2023). *Can LLM Already Serve as a Database Interface? A Big Bench for Large-Scale Database Grounded Text-to-SQLs (BIRD)*. NeurIPS. https://arxiv.org/abs/2305.03111. ↩

.4. Vanna AI. (2026). *Vanna 2.0: Turn Questions into Data Insights*. GitHub. Consultado el 26 de mayo de 2026. El repositorio público consultado aparece como archivado el 29 de marzo de 2026. ↩