# Propuesta de Arquitectura (50M registros, miles de QPS)

### Contexto

El sistema debe escalar hasta **50 millones de registros** y soportar **miles de consultas por segundo**, manteniendo una latencia baja y resultados consistentes. El objetivo principal es resolver búsquedas aproximadas de nombres, tolerando errores tipográficos, variantes y diferencias de escritura.

Se proponen **dos escenarios complementarios**, uno basado en ingeniería de datos clásica y otro con un enfoque AI-Native, ambos funcionales para un entorno productivo.

Si bien hoy en día en la industria hay amplia variedad de stack tecnológico disponible para arquitecturas de big data, prioricé utilizar herramientas sobre las cuales tenga conocimientos por haberlas utilizado con anterioridad. Además busqué dentro de lo posible mantener la “simpleza” en el diseño, teniendo en cuenta que ante dos arquitecturas igual de robustas y funcionales para el mismo servicio, es preferible la más simple que requiera menor complejidad en el mantenimiento y sobre todo, más económica.

## Premisas de diseño (comunes a ambos escenarios)

- **Search engine ≠ Source of Truth (SoT)**: el índice acelera búsquedas, pero los datos “verdaderos” viven en una base primaria.
- **Búsqueda en 2 etapas**:
    1. *Candidate retrieval* (rápido) → reduce de 50M a cientos.
    2. *Scoring / ranking* (más caro) → aplica solo a ese subconjunto.
- **Read/Write desacoplados** Hoy en día en la industria se conoce como patrones CQRS pero al fin y al cabo es un pipeline asíncrono de indexación donde desacoplamos la lectura de la escritura para mejorar performance.
- **Cache** para “hot queries” y para amortiguar picos.
- Medición: p50/p95 de latencia, tasa de cache hit, y tamaño medio del set de candidatos.

---

# Escenario A — Standard (Data Engineering clásico)

# Escenario A: Arquitectura Standard (Ingeniería de Datos Clásica)

En este escenario, la búsqueda se apoya en un **motor de búsqueda textual especializado**, mientras que la persistencia y la consistencia de los datos se delegan a una **base de datos primaria** que actúa como *Source of Truth*. La idea es diferenciar el índice de búsqueda de la base de datos para evitar problemas en los datos por corrupción del índice. Por simpleza definimos una PostgreSQL bien particionada y replicada pero la opción premium sería una base de datos columnar como Cassandra.

La API es completamente **stateless**, lo que permite escalar horizontalmente. Ante una consulta, el flujo típico es:

1. Normalizar el nombre de entrada.
2. Consultar Redis para resolver búsquedas frecuentes.
3. Consultar Elasticsearch/OpenSearch para obtener un conjunto reducido de candidatos (IDs + score).
4. Recuperar los datos completos desde la base primaria.
5. Ordenar y devolver los resultados.

La indexación se realiza de forma **asíncrona** mediante pipelines de ingestión. Aquí puede incorporarse un Kafka que encole para que luego Spark los procese en micro batches para no saturar el indexado si llegan de a 100k registros. Las escrituras impactan primero en la base de datos y luego en el índice de búsqueda, aceptando una consistencia eventual que es razonable para este tipo de problema.

Para tolerar errores tipográficos y variaciones, se utilizan **character n-grams**, normalización de texto y un esquema de scoring explicable. Esto permite un buen equilibrio entre performance, precisión y control del sistema. Si bien podría usarse el fuzzy estándar de ElasticSearch u OpenSearch, al escalar se vuelve muy costoso.  Incorporamos una capa de cache en Redis donde se cachean no solo resultados, sino también **tokens fonéticos** comunes para evitar re-calcular hashings costosos

## Objetivo

Búsqueda textual tolerante a errores, con latencia baja y operación sencilla.

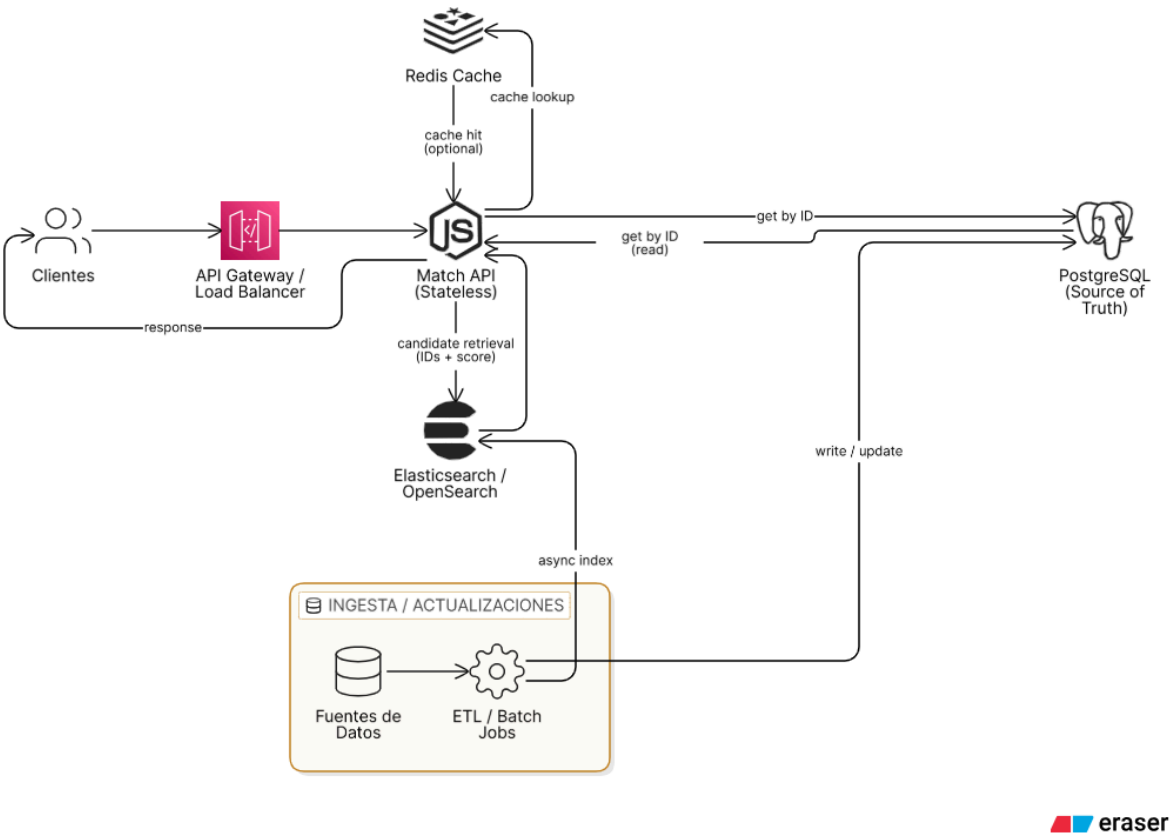
## Componentes

- **API (stateless)**: FastAPI/Go en contenedores (K8s o autoscaling managed).
- **SoT (base primaria)**: PostgreSQL (o Cassandra) + particionado/índices, o un store gestionado equivalente si es cloud.
    - En este caso, el patrón de acceso típico es **“get by id”** (una vez que el motor de búsqueda devuelve IDs).
- **Motor de búsqueda**: Elasticsearch/OpenSearch para candidate retrieval.
- **Cache**: Redis (look-aside).

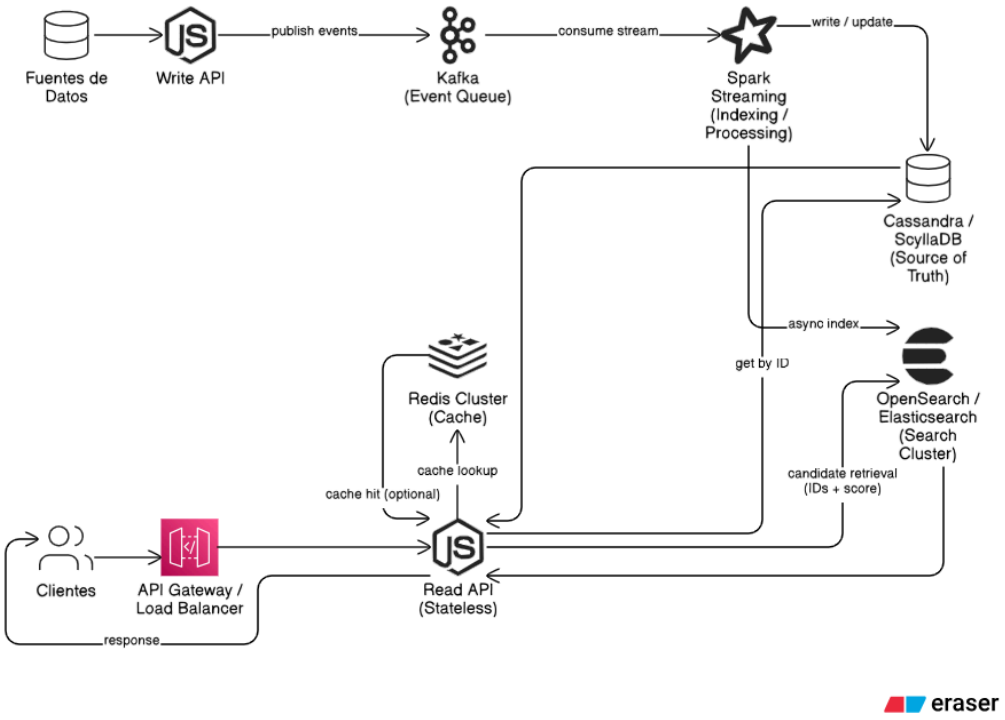
- **Ingesta y procesamiento**: pipeline batch/stream (Kafka + Spark opcional; para empezar puede ser un job incremental + colas).

## Diagrama

Arquitectura Base




Arquitectura premium




## Cómo lograr baja latencia en búsquedas de texto

- Indexo en Elastic un campo `name_normalized` con:
  - lowercase + asciifolding (acentos),
  - limpieza de puntuación,
  - **character n-grams** (tolerancia a typos).
- Evito un `fuzzy` “abierto” en todo el índice (caro).
  En su lugar:
  - n-grams + boosting por tokens,
  - y fuzzy acotado (como fallback y con límites) si hace falta.

  Elastic devuelve **IDs y un score**, no “todo el documento pesado”. El detalle completo lo saco del SoT por id.

## Indexación y crecimiento exponencial

- **Shards + replicas** según volumen.
- **Reindex con alias** para cambios de analyzer sin downtime.
- Pipeline asíncrono: las escrituras van primero al SoT y luego al índice (eventual consistency aceptable para búsqueda).

## Cache y disponibilidad

- Redis cachea:
  - resultados top-k por query normalizada (TTL corto),

    - opcional: embeddings/tokenizaciones precomputadas si aplica.
- Alta disponibilidad:
    - API con múltiples réplicas,
    - Elastic con replicas, snapshot/restore,
    - DB con réplicas de lectura si la hidratación por id crece mucho.

**Pros**

- Latencia muy baja y costo razonable.
- Tecnología madura y defendible.
- Matching explicable (ideal para challenge).

**Contras**

- Menos “inteligencia” ante variaciones culturales complejas.
- Requiere tuning de analyzers y relevancia.

---

# Escenario B — AI-Native (Vector + re-ranking controlado)

El segundo escenario propone un núcleo basado en **representaciones vectoriales**, pensado para mejorar el matching en casos más complejos (variantes culturales, transliteraciones, alias).

En lugar de depender únicamente del texto, los nombres se transforman en **embeddings** que capturan similitudes a nivel de caracteres y sub-palabras. Estos vectores se almacenan en una base vectorial optimizada para búsquedas aproximadas (ANN). Una mejor opción para nombres es usar Hybrid Search que combina el uso de vectores con score de palabras clave.

El flujo de búsqueda es:

1. Generar el embedding del nombre de entrada.
2. Recuperar los Top-K candidatos desde la base vectorial.
3. Aplicar un **re-ranking liviano** solo si el resultado no es concluyente.
4. Recuperar los datos finales desde la base primaria.

El re-ranking no se ejecuta siempre, sino de manera selectiva, para evitar penalizar la latencia. Además, este enfoque puede combinarse con señales textuales (búsqueda híbrida) para mejorar la precisión en coincidencias exactas.

Este escenario introduce mayor complejidad operativa, pero permite evolucionar el sistema hacia un enfoque más inteligente y adaptable, especialmente cuando el matching basado en reglas empieza a mostrar limitaciones.

## Objetivo

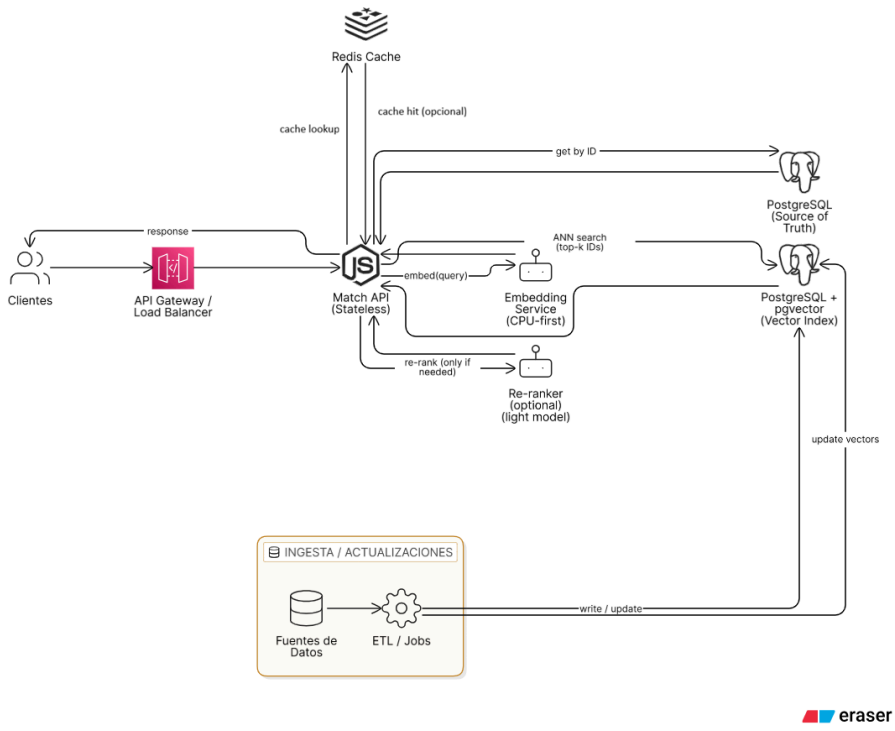
Mejor recall/precisión cuando hay mucha variación (transliteración, errores sistemáticos, alias), evitando reglas manuales.

## Componentes

- **API stateless**.
- **Embedding service** (CPU-first; GPU opcional).
- **Vector DB / ANN index**: Si la exigencia es simple pgvector (Postgres) es la alternativa más económica, que si bien va a consumir mucha RAM nos ahorra tener que utilizar un nuevo sistema. En caso de mayor exigencia Qdrant/Milvus soportan la búsqueda híbrida que definimos como alternativa.
- **Re-ranking**: modelo liviano y **selectivo** (no siempre).
- **SoT**: misma DB primaria del escenario A.

## Diagrama

Arquitectura Base




## Representación (embeddings)

En nombres propios, un modelo BERT genérico fallaría por lo que lo mejor sería poder hacer un **Bi-encoder** entrenado/ajustado en datos de pares (match/no match). Como alternativa si no hay dataset de entrenamiento podemos implementar embeddings basados en **subword/character** como **CharacterBERT** o un modelo liviano preentrenado y validado con pruebas.

### Búsqueda vectorial

- ANN (HNSW) para devolver top-K candidatos (ej. 100).
- Filtrado por metadatos si existieran (país, tipo de entidad, etc.).

### Re-ranking sin matar latencia

- Re-ranking **solo si hace falta**:
    - si el top-1 tiene mucha ventaja, no re-ranqueo,
    - si hay empate o baja confianza, re-ranqueo top-20/top-50.
- Re-ranker liviano (distilled / ONNX en CPU) o cross-encoder pequeño.

### Hybrid (opcional)

Como dijimos en la búsqueda vectorial  podemos combinar vector score + score léxico (BM25) usando Qdrant o Milvus soportan este tipo de búsquedas.

**Pros**

- Mejor recall en casos difíciles.
- Menos reglas manuales.
- Evoluciona mejor con datos (aprende de feedback).

**Contras**

- Más complejidad y costo que el enfoque clásico.
- Operación de modelos (versionado, drift).
- Si se re-ranquea demasiado, sube latencia.

---

# Conclusión

Ambas arquitecturas son complementarias.
Una estrategia realista sería **comenzar con el enfoque standard**, validando performance y calidad, y **evolucionar hacia un enfoque AI-Native** solo cuando los datos y los requerimientos lo justifiquen.

Esta decisión minimiza riesgos, evita sobre-ingeniería y permite escalar el sistema de forma progresiva y controlada.